How to Contact dSPACE

Mail: dSPACE GmbH
     Technologypark 25
     33100 Paderborn
     Germany

Tel.: +49 5251 1638-0
Fax: +49 5251 66529
E-mail: info@dspace.de
Web: http://www.dspace.de
Technical Support: support@dspace.de
     +49 5251 1638-941
     http://www.dspace.de/goto?support

How to Contact dSPACE Support

dSPACE recommends that you use dSPACE Support Wizard to contact dSPACE support.
It is available
• On your dSPACE CD at \Diag\Tools\dSPACESupportWizard.exe
• Via Start – Programs – dSPACE Tools (after installation of the dSPACE software)
• At http://www.dspace.de/goto?supportwizard
   You can always find the latest version of dSPACE Support Wizard here.

Software Updates and Patches

dSPACE strongly recommends that you download and install the most recent patches for
your current dSPACE installation. Visit http://www.dspace.de/goto?support for software
updates and patches.

Important Notice

This document contains proprietary information that is protected by copyright. All rights
are reserved. Neither the documentation nor software may be copied, photocopied,
reproduced, translated, or reduced to any electronic medium or machine-readable form, in
whole or in part, without the prior written consent of dSPACE GmbH.

© Copyright 2005 by:
dSPACE GmbH
Technologypark 25
33100 Paderborn
Germany

This publication and the contents hereof are subject to change without notice.
Brand names or product names are trademarks or registered trademarks of their respective
companies or organizations.
## Contents

### Documentation Overview
- Documentation Types ................................................................. 9
- About This Guide ........................................................................... 10
- Related Documents ......................................................................... 12

### Introduction to RTI and RTI-MP
- Development Steps ................................................................. 17
- Throttle Control Example ............................................................... 18
- Using the RTI and RTI-MP Implementation Guide ......................... 20

### Working with RTI and RTI-MP
- How to Activate a Specific Platform Support ............................ 24
- How to Access RTI Blocks ............................................................... 25
- How to Customize the MATLAB Start-Up .................................. 26
- How to Set Default Values for Build Options ............................. 27
- How to Specify RTI-Specific Settings for a Model ....................... 28
- How to Remove the RTI-Specific Settings from a Model ............... 29

### Defining I/O Interfaces
- Naming Conventions ................................................................... 30
- PHS-Bus Address and Board Number .......................................... 31
- How to Add I/O Blocks ................................................................. 32
- Vectorized Versus Scalar I/O Blocks ........................................... 33
- Data Typing Basics ...................................................................... 34
- How to Enable Data Typing .......................................................... 35

### Handling Tasks
- Real-Time Kernel and Scheduler .................................................. 36
- Priorities and Task-Switching Time .............................................. 37
- Task States and Execution Order ............................................... 38
- Overrun Situation and Turnaround Time .................................... 39
- Timer Tasks ................................................................................. 40
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer Task Basics</td>
<td>64</td>
</tr>
<tr>
<td>Single Timer Task Mode</td>
<td>65</td>
</tr>
<tr>
<td>Multiple Timer Task Mode</td>
<td>66</td>
</tr>
<tr>
<td>Comparing the Execution Modes</td>
<td>73</td>
</tr>
<tr>
<td>How to Bind a Timer Task Explicitly to an Interrupt Block</td>
<td>73</td>
</tr>
<tr>
<td>Tasks Driven by Interrupt Blocks</td>
<td>75</td>
</tr>
<tr>
<td>Types of Triggered Subsystems</td>
<td>76</td>
</tr>
<tr>
<td>Preparing a Function-Call Subsystem</td>
<td>78</td>
</tr>
<tr>
<td>How to Use Hardware Interrupts</td>
<td>87</td>
</tr>
<tr>
<td>How to Use Software Interrupts</td>
<td>89</td>
</tr>
<tr>
<td>Example of a Task Driven by a Hardware Interrupt Block</td>
<td>91</td>
</tr>
<tr>
<td>Assigning Parts of the Model to the Background Task</td>
<td>93</td>
</tr>
<tr>
<td>Basics of the Background Task</td>
<td>93</td>
</tr>
<tr>
<td>How to Assign Parts of the Model to the Background Task</td>
<td>94</td>
</tr>
<tr>
<td>Example of a Subsystem Assigned to the Background Task</td>
<td>95</td>
</tr>
<tr>
<td>How to Change the Task Priorities</td>
<td>96</td>
</tr>
<tr>
<td>How to Handle Overrun Situations</td>
<td>99</td>
</tr>
<tr>
<td>How to Combine Several Interrupt Sources</td>
<td>101</td>
</tr>
<tr>
<td>How to Subschedule Tasks</td>
<td>104</td>
</tr>
<tr>
<td>Basics of Time-Triggered Tasks and Timetables</td>
<td>106</td>
</tr>
<tr>
<td>How to Implement Time-Triggered Tasks</td>
<td>110</td>
</tr>
<tr>
<td>How to Implement Timetables</td>
<td>113</td>
</tr>
<tr>
<td>Distributing the Model for MP Systems</td>
<td>117</td>
</tr>
<tr>
<td>How to Configure an RTI-MP Model</td>
<td>120</td>
</tr>
<tr>
<td>Standard Network Topology</td>
<td>122</td>
</tr>
<tr>
<td>Interprocessor Communication</td>
<td>123</td>
</tr>
<tr>
<td>GigaLink Connection</td>
<td>124</td>
</tr>
<tr>
<td>Swinging Buffer Protocol</td>
<td>125</td>
</tr>
<tr>
<td>Virtual Shared Memory Protocol</td>
<td>128</td>
</tr>
<tr>
<td>How to Implement Interprocessor Communication</td>
<td>129</td>
</tr>
<tr>
<td>Model Evaluation</td>
<td>133</td>
</tr>
<tr>
<td>Communication with a Sample Time Transition</td>
<td>137</td>
</tr>
<tr>
<td>How to Define a Default CPU</td>
<td>145</td>
</tr>
<tr>
<td>Interprocessor Interrupts</td>
<td>147</td>
</tr>
<tr>
<td>How to Implement Interprocessor Interrupts</td>
<td>147</td>
</tr>
<tr>
<td>How to Change the Task Priorities in RTI-MP Models</td>
<td>149</td>
</tr>
<tr>
<td>Model Distribution Guidelines</td>
<td>151</td>
</tr>
</tbody>
</table>
Contents

Inserting Custom C Code ............................................................... 153
Implementing S-Functions ............................................................... 156
Basics of S-Functions ................................................................... 156
Handcrafting the C-Coded S-Function ............................................ 158
Implementing User-Code ............................................................... 164
How to Use User-Code in Simulink Models ................................. 164
How to Generate a Template for the User-Code ......................... 167
Handcrafting the User-Code ......................................................... 168
Comparing S-Functions and User-Code ........................................ 174
Execution Order of S-Functions and User-Code .......................... 175
Tips and Tricks for Custom C Code .............................................. 177
Flow Chart for the Custom C Code of the Throttle Control Example ............................................................... 178
C Code for the Throttle Control Example's S-Function ............... 179
C Code for the Throttle Control Example's User-Code ............... 182
How to Access Custom Variables via ControlDesk ..................... 184
How to Handle Application-Specific Errors ................................ 186
Using Slave Applications ............................................................. 188
Example of a Slave Application as User-Code ............................ 188
Example of a Slave Application as an S-Function ........................ 191

Managing the Simulation .............................................................. 193
Simulation Control (RUN/STOP Mechanism) ............................... 195
How to Set and Read the Simulation State .................................. 197
Time-Stamping and Data Acquisition ......................................... 200
How to Modify Data Acquisition Behavior .................................. 205
Reaction to Run-Time Errors ...................................................... 208
Different Results from Different Simulation Modes .................... 209

Advanced Techniques ................................................................. 211
Optimizing the Build Process and Simulation via Atomic Subsystems ....................................................................... 213
Virtual Versus Atomic Subsystems ............................................. 213
How to Bundle the Execution of Blocks ..................................... 214
How to Reduce the Function and File Sizes via Atomic Subsystems ....................................................................... 216
Excluding Subsystems from the TRC File .................................... 218
Advantages of a Reduced TRC File ............................................ 218
How to Exclude a Subsystem from the TRC File ........................... 220
How to Apply Subsystem Permissions to the TRC File ................. 221
Modifying the Block Execution Order........................................... 223
Determining the Execution Order.............................................. 223
How to Check for Illogical Block Priorities................................. 225
Tuning Parameters of Transfer Fcn, Zero-Pole and State-Space Blocks.................................................................................................................. 226
MATLAB's State-Space Representation for a Transfer Function... 227
How to Tune Transfer Function Parameters via Manual Conversion.................................................................................................................. 229
Tuning Transfer Function Parameters via External Simulation.... 229
How to Tune Transfer Function Parameters via Parameter Inlining.................................................................................................................. 230
How to Keep Zero Parameters of State-Space Blocks .............. 231
Tuning Parameters of n-D Look-Up Table Blocks ................... 232
How to Migrate from AlphaCombo to DS1005 or DS1006......... 234
How to Use Simulink's Model Verification Blocks ...................... 238

Building and Downloading the Model ......................................... 241
Basics for the Build and Download............................................. 243
How to Specify Options for the Build Process............................ 245
How to Start the Build and Download........................................ 248
How to Automate the Build Procedure...................................... 250
How to Update TRC and SDF Files........................................... 253
How to Download DS230x Applications .................................. 254
Debugging an Application.......................................................... 256
How to Run the RTI-MP Diagnostic Tool................................. 256
Handling Exceptions..................................................................... 258
How to Create an Application for a Virtual dSPACE System....... 265
Adding User-Specific Files, Source Files and Libraries ............ 266

External Simulation .................................................................. 267
How to Run an External Simulation........................................... 269

Conventions and Limitations ...................................................... 271
Supported Toolboxes and Blocksets from The MathWorks .......... 272
Data Typing Restrictions.............................................................. 273
Conventions .............................................................................. 274
Limitations.................................................................................. 275
General Limitations ................................................................. 275
Contents

Limitations with RTI-MP ............................................................ 279
Limitations with Stateflow ........................................................ 280
Limitations for Custom C Code................................................. 280

Troubleshooting ................................................................. 283
Implementing the Model........................................................... 285
Automatic Program Building.................................................... 289
Checking the Model ............................................................... 291
Generating the Code .............................................................. 293
Compiling and Linking the Code.............................................. 294
Initializing the Application...................................................... 296
Running the Application ......................................................... 297
Running ControlDesk.............................................................. 300

Index ................................................................. 303
Documentation Overview

The dSPACE documentation consists of different types of documents, see *Documentation Types* on page 10.

For information about the contents of the *RTI and RTI-MP Implementation Guide*, see *About This Guide* on page 12.

For information on the documents that are available for Real-Time Interface for single-processor (RTI) and multiprocessor (RTI-MP) dSPACE systems, see *Related Documents* on page 14.
Documentation Overview

Documentation Types

After you install your dSPACE system, you can access the entire documentation as online help or printable Adobe® PDF files. You will also receive a printed version of some important documents.

dSPACE HelpDesk

dSPACE HelpDesk is your primary source of information on both the hardware and the software of your dSPACE system.

To open dSPACE HelpDesk

➤ Select dSPACE HelpDesk from the dSPACE Tools program group of the Windows Start menu.

From each dSPACE HelpDesk page, you can easily search and navigate to the desired information. You also have direct access to printable Adobe PDF files: see How to Work with dSPACE HelpDesk in dSPACE HelpDesk.
Only the documents of the products installed on your system are available. The entire product documentation is available if you open dSPACE HelpDesk on the dSPACE CD.

**dSPACE HelpDesk structure**

The structure of the documents in dSPACE HelpDesk reflects the different phases of your work:

- Installation and Configuration
- Implementation
- Experiment and Test
- Production Code Generation
- Calibration

The topics that are shown depend on your dSPACE system.

**Context-sensitive help**

When you work with any dSPACE software, you can get context-sensitive help via the F1 key and/or Help button.

**PDF Files**

All documents are also available as printable Adobe PDF files in the %DSPACE_ROOT%\Doc\Print folder. see *How to Work with dSPACE HelpDesk* in dSPACE HelpDesk.

**Printed Documents**

You will receive a printed version of the documents that are essential for working away from your PC.
About This Guide

This guide introduces you to Real-Time Interface (RTI and RTI-MP) for dSPACE systems. RTI acts as the link between Simulink®, Real-Time Workshop®, and the dSPACE hardware. RTI-MP enables you to partition your system model and allocate the parts to the different CPUs of a multiprocessor system.

After reading through this guide you will be able to use RTI and RTI-MP to add I/O interfaces, interrupts, etc. to your Simulink model, build real-time code, and download and execute it on dSPACE hardware.

There are a few limitations concerning RTI’s support for Simulink blocks. For details, refer to Conventions and Limitations on page 271.

For a brief overview of the new features of RTI and RTI-MP, refer to Key Features of dSPACE Release 4.2 in dSPACE Release New Features and Migration.

- dSPACE Release 4.2 supports two MATLAB releases. This guide describes the MATLAB R13.x support in dSPACE Release 4.2. If you install the MATLAB R14SP1 support in dSPACE Release 4.2, you must consider additional features and limitations. Refer to Key Features of the MATLAB R14SP1 Support in dSPACE Release 4.2 and Limitations of the MATLAB R14SP1 Support in dSPACE Release 4.2 in dSPACE Release New Features and Migration.

- This guide describes Real-Time Interface for all hardware platforms. Not all sections apply to all platforms. You can skip all paragraphs that are not relevant to your platform.

- Where you find terms like rti<xxxx> in this guide (for example, in path or file names), replace them by the RTI platform support you are using, for example, rti1005.
Where you find terms like `<model>` or `<submodel>` in this guide, replace them by the actual name of your model or submodel. For example, if the name of your Simulink model is `smdl103_sl.mdl` and you are asked to edit the `<model>_user.c` file, you actually have to edit the `smdl103_sl_user.c` file.

Legend

The following symbols are used in this document.

- **Warnings**: provide indispensable information to avoid severe damage to your system and/or your work.

- **Notes**: provide important information that should be kept in mind.

- **Tips**: show alternative and/or easier work methods.

- **Examples**: illustrate work methods and basic concepts, or provide ready-to-use templates.
Related Documents

Below is a list of documents that you are also recommended to read when working with RTI and RTI-MP.

- **RTI and RTI-MP**
  - *RTI and RTI-MP Implementation Guide* gives detailed information and instructions on using Real-Time Interface (RTI and RTI-MP) to implement your real-time models.
  - *RTI and RTI-MP Implementation Reference* offers reference information on the various dialogs, files, options, etc. of Real-Time Interface (RTI and RTI-MP) for dSPACE systems. It also describes the blocks introduced by RTI-MP.
  - The board-specific *RTI references* provide concise information on the board's RTI library.

- **RTLib**
  - The board-specific *RTLib references* provide detailed descriptions of the C functions needed to program RTI-specific Simulink S-functions or implement your control models manually via C programs (handcoding).

- **Features**
  - The board-specific *Features documents* (DS<xxxx> Features or DS<xxxx> User's Guide) provide feature-oriented access to the information you need to implement your control models on your real-time hardware.

- **Hardware**
  - The board-specific *hardware references* provide information on the board's hardware characteristics.

- **Processor and controller boards**
  - *DS1005 Implementation Documents, DS1006 Implementation Documents, DS1103 Implementation Documents, DS1104 Implementation Documents, MicroAutoBox Implementation Documents* provide information on the features, RTI, RTLib and the hardware of the processor and controller boards.

- **Connectable I/O boards**
  - *I/O Board Implementation Reference* provides detailed information on the features, RTI, RTLib and hardware of the dSPACE I/O boards that can be connected to a dSPACE processor board.
First work steps

- dSPACE System First Work Steps provides information on configuring dSPACE systems after you installed the dSPACE hardware. It shows you how to get started with your dSPACE system after installation. This document is aimed at users who have no experience with dSPACE systems.

ControlDesk Standard

- ControlDesk Experiment Guide introduces you to the experiment features provided by ControlDesk Standard.
- ControlDesk Automation Guide shows you how to automate the features provided by ControlDesk Standard.
- ControlDesk Reference, ControlDesk Instrument Reference and ControlDesk Automation Reference provide detailed information on the menus, context menus, dialogs and Python libraries contained in ControlDesk Standard.

AutomationDesk

- AutomationDesk Guide introduces you to the automation features provided by AutomationDesk.
- AutomationDesk Tutorial has several lessons that guide you through using AutomationDesk.
- AutomationDesk Reference provides detailed information on the menus, context menus, and dialogs contained in AutomationDesk.
- AutomationDesk Library Reference provides detailed information on the libraries supported by AutomationDesk.

Interface libraries

- MLIB/MTRACE MATLAB-dSPACE Interface Libraries contains detailed reference information and instructions on the experiment features of MLIB/MTRACE.
- CLIB C Interface Library contains detailed reference information on the C Interface Library CLIB, which contains C functions to access the dSPACE processor and controller boards from the host PC.
Introduction to RTI and RTI-MP

Developing a controller
In control engineering hardly any controller is designed without simulation. The typical procedure is to develop a model of the plant and simulate it on a computer. Then a controller is added to the simulation and optimized. Refer to Development Steps on page 18.

Example
The throttle control example demonstrates these steps from Simulink to real-time simulation for MicroAutoBox 1401/1501. Refer to Throttle Control Example on page 20.

Using this guide
This typical procedure from Simulink to real-time simulation is also reflected in the structure of this implementation guide. Refer to Using the RTI and RTI-MP Implementation Guide on page 24.
Development Steps

Control design and Simulink simulation

At this stage of development the simulation is performed in Simulink. The major feature of this type of simulation is that the computer has as much time as needed to calculate the behavior of the system. Thus, if your model is simple, the results can be calculated quickly. If your model is complex, much more time is needed to carry out the necessary calculations. However, because you do not need to fulfill a strict time requirement for Simulink simulation, you do not need to reduce the complexity of the model.

RCP with dSPACE Prototyper

The whole situation is different once the simulated control system meets your expectations and you want to test the controller on the actual plant. Since the controller might need further modifications, you do not want to produce it in hardware at this stage of development. Therefore, you have to connect the real plant to a controller that is simulated in real time – this technique is called rapid control prototyping (RCP). The major feature of this real-time simulation is that the simulation has to be carried out as quickly as the real system would actually run, thereby allowing you to combine the simulation and the real plant.

HIL with dSPACE Simulator

When your simulated controller is able to control your real plant, you typically produce the actual controller. For the final tests you usually connect the real controller to a model of the plant, which, of course, has to be simulated in real-time. This way you can ensure that the controller does not contain any errors that could damage the real plant. This technique is called hardware-in-the-loop simulation (HIL).

For both RCP and HIL the real-time simulation is rather important. The computing power required by real-time simulation highly depends on the characteristics of the simulated model: If it contains very demanding calculations you have to provide a lot of computing power because the timing cannot be satisfied otherwise. dSPACE systems fulfill this demand for computing power.
Because real-time simulation is such a vital aspect for control engineering, the same is true for the automatic generation of real-time code, which can then be implemented on the hardware. For dSPACE systems, Real-Time Interface (RTI/RTI-MP) carries out this linking function. Together with Real-Time Workshop from The MathWorks it automatically generates real-time code from Simulink models and implements this code on dSPACE real-time hardware. Therefore, you save time and effort twice:

- You do not need to manually convert your Simulink model into another language such as C.
- You do not need to be concerned about a real-time program frame and I/O function calls, or about implementing and downloading the code onto your dSPACE hardware.

RTI carries out these steps for you. You just have to add the required dSPACE blocks (for example, I/O interfaces) to your Simulink model.
Throttle Control Example

The following example demonstrates the steps from Simulink to real-time simulation for MicroAutoBox 1401/1501.

Control design and Simulink simulation

Suppose you want to develop an electronic throttle control (ETC or drive-by-wire) for a throttle valve like the one shown below.

ETC departs from conventional mechanical systems by replacing the Bowden cable or linkage with an ECU and the electric motor for throttle valve control. The system can thus control and modify the throttle valve aperture with reference to numerous operating parameters, performing tasks such as torque reduction for electronic traction control.
The first step in developing an ETC is to model the throttle valve itself. Then you have to add the model of the actual electronic throttle control. You can use the resulting Simulink model to optimize the ETC in Simulink simulation. An appropriate Simulink model is shown in the illustration below:
Rapid control prototyping

The next step is to build the real-time code for the ETC (including I/O interfaces) and to download it onto the dSPACE hardware. You can then connect it to the real throttle valve for further optimization (rapid control prototyping). The following illustration shows the arrangement according to the MicroAutoBox 1401/1501 demonstration system:

Hardware-in-the-loop simulation

If you want to control the simulated throttle valve at a later stage with the ETC (hardware-in-the-loop simulation), the arrangement is the other way round.

This guide will show how to implement I/O interfaces in your models below.

Build & download

For a single-processor system such as MicroAutoBox 1401/1501, the following steps are necessary to generate real-time code for the model, to download the code onto the dSPACE hardware and to simulate it. To carry out these steps you have to open the Simulation Parameters dialog via the Simulation – Parameters menu command.
Choose a fixed-step solver and specify a fixed-step size on the Solver page of the Simulation Parameters dialog.

- The Start time has to be zero.
- “Open end” simulation via \( \infty \).
- Fixed-step solver only.
- Desired step size for the real-time simulation.
- Desired timer task mode.

Specify the correct settings for the Real-Time Workshop page for MicroAutoBox 1401/1501 (select the “Target configuration” category).

The various categories let you configure options for code generation and the make process.

- The Build button starts the build and download.

When you start the build and download procedure, Real-Time Workshop and RTI generate the real-time code and download it to your dSPACE hardware – provided that the hardware is registered correctly with the Platform Manager of ControlDesk.
Using the RTI and RTI-MP Implementation Guide

Now you are able to convert a Simulink simulation into a real-time simulation running on dSPACE hardware. However, you still need to know in detail how to carry out the various steps of implementing dSPACE components into your Simulink models, which is the subject of this guide. The typical procedure from Simulink simulation to real-time simulation is mirrored by the structure of this implementation guide.

**RTI and RTI-MP environment**

RTI supports all dSPACE platforms. If you selected more than one platform during installation, you can choose which platform to activate. You cannot work with more than one platform at a time. Simulink models can be migrated between the different platforms. Refer to *Working with RTI and RTI-MP* on page 27.

**I/O interfaces**

To connect the real-time simulation to the physical world, you need to introduce I/O interfaces into your model. Refer to *Defining I/O Interfaces* on page 39.

**Tasks**

Your Simulink model usually consists of parts that have to be calculated periodically and others that have to be calculated at aperiodic points in time. Refer to *Handling Tasks* on page 51.

**RTI-MP models**

If the computing power of single-processor systems does not meet your requirements you can use a dSPACE multiprocessor system. Refer to *Distributing the Model for MP Systems* on page 117.

**Custom C code**

S-functions are a feature of Simulink that allow you to implement user-defined functionality. User-Code is a similar feature, but is provided by RTI. Refer to *Inserting Custom C Code* on page 153.

**Simulation management**

Managing the simulation covers simulation control, data acquisition, and error handling. Refer to *Managing the Simulation* on page 193.
Advanced techniques  You can optimize the simulation via atomic subsystems and block priorities. Parameter tuning for Transfer Fcn and Zero-Pole blocks can be enabled by one of three methods. These and other tips and tricks can be found in Advanced Techniques on page 211.

Build and download  When an RTI or RTI-MP model is complete, real-time code can be generated and downloaded to the dSPACE system. Refer to Building and Downloading the Model on page 241.

External simulation  The external mode is a simulation mode provided by Simulink, Real-Time Workshop and RTI. This mode supports on-the-fly parameter tuning in a real-time environment. Refer to External Simulation on page 267.

Limitations  Even though RTI is designed to work seamlessly with Simulink and Real-Time Workshop, there are a few technically related conventions and limitations. Refer to Conventions and Limitations on page 271.

Troubleshooting  When implementing an RTI or RTI-MP model, when building and downloading the real-time application or when running the simulation, you might experience problems. If any RTI-related problem occurs, follow the instructions provided by the error message. Refer to Troubleshooting on page 283.

Related Topics

New features  For a brief overview of RTI's and RTI-MP's new features, refer to Key Features of dSPACE Release 4.2 in dSPACE Release New Features and Migration.
Working with RTI and RTI-MP

The following aspects are important when you start working with RTI and RTI-MP.

Selecting the platform
If you selected more than one platform during installation, you can choose which platform to activate. You cannot work with more than one platform at a time. Refer to How to Activate a Specific Platform Support on page 29.

Accessing RTI blocks
There are two ways to access the RTI and RTI-MP blocks. Refer to How to Access RTI Blocks on page 30.

Customizing the environment
You can make MATLAB® run custom scripts on start-up. Refer to How to Customize the MATLAB Start-Up on page 33.
You might want to customize the simulation environment. Refer to How to Set Default Values for Build Options on page 34.
Migrating a model

You might want to migrate a model to a different RTI platform support. Refer to How to Specify RTI-Specific Settings for a Model on page 37.

You might want to remove all the RTI-specific settings from a model so that it can be opened in a MATLAB/Simulink environment without RTI installation. Refer to How to Remove the RTI-Specific Settings from a Model on page 38.
How to Activate a Specific Platform Support

If you selected more than one platform during installation, you can choose which platform to activate. You cannot work with more than one platform at a time.

To start RTI for a particular platform

➤ When MATLAB is started, RTI displays a dialog that lets you choose the desired platform.

If you select the Do not show this dialog again checkbox, MATLAB will always start with the platform that was last active.

To activate a different platform

➤ Change to the MATLAB prompt and type rti<XXXX> to activate the desired platform (replace rti<XXXX> with the platform you want to use: for example, rti1103).

The corresponding RTI library opens automatically.
How to Access RTI Blocks

The various RTI and RTI-MP blocks allow you to access the dSPACE hardware.

You must not break the library links of the RTI blocks. Otherwise, the blocks cannot be updated when you migrate your model to a new version of RTI and it is very likely that your model will not operate correctly under such a condition.

There are two ways to get a specific block: You can use the Simulink Library Browser, or the RTI and RTI-MP block libraries.

The MATLAB Launch Pad provides shortcuts to the various block libraries, demo libraries, the online help and also information on the Internet.
To access an RTI block via the Simulink Library Browser

1. Open the Simulink Library Browser.
2. Expand the relevant dSPACE library, and find the block in one of the sublibraries it contains.
3. Drag and drop the block into your model.

The following illustration shows the dSPACE RTI1103 library in the Simulink Library Browser:

To access an RTI block via the RTI library

1. Open the RTI library by typing `rti` at the MATLAB prompt, open the corresponding sublibrary, and find the block.

To access multiprocessor-specific blocks you have to open the RTI-MP library by typing `rtimp` at the MATLAB prompt.
If you want to use only a single processor board of a multiprocessor system, you do not need to add any RTI-MP-specific blocks to your model.

2. Drag and drop the block into your model.

The following illustration shows the dSPACE RTI1103 library:
How to Customize the MATLAB Start-Up

You can make MATLAB run custom scripts on start-up, which are invoked either before or after the dSPACE initialization.

To invoke a custom script before dSPACE initialization
➤ Add the necessary commands in the startup.m file. This has to be placed in the MATLAB search path.

The startup.m file is a standard MATLAB feature. For further information, type doc startup at the MATLAB prompt. See also startup.m in the RTI and RTI-MP Implementation Reference.

To invoke a custom script after dSPACE initialization
➤ Add the necessary commands in the dsstartup.m file, for example, to automatically open a library or model whenever MATLAB is started. The dsstartup.m file has to be placed in the MATLAB search path.

For details, see dsstartup.m in the RTI and RTI-MP Implementation Reference.

Suppose you want MATLAB to automatically change to your working folder C:\Work and open the Simulink model my_model.mdl. Write the following dsstartup.m file:

cd C:\Work
my_model
How to Set Default Values for Build Options

Whenever you create a new RTI model, its build and simulation parameters are set to default values. For certain parameters and options (such as the solver settings) you can define the default values via Simulink's Preferences dialog.

**RTI**  The initial default values, which are set by the MATLAB installation, are not suitable for building real-time applications with RTI.

**RTI-MP**  The default values specified in Simulink’s Preferences dialog do not affect the simulation parameters of RTI-MP models.

**To specify default values for the simulation parameters**

1. Select **Preferences** from the **File** menu of the MATLAB Desktop.
2. Expand the **Simulink** node in the Preferences tree, and select the **Simulation** item.
Specify the desired settings. For building real-time applications with RTI, you should specify the following defaults on the Solver page:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation time frame</td>
<td></td>
</tr>
<tr>
<td>Start time</td>
<td>0.0</td>
</tr>
<tr>
<td>Stop time</td>
<td>inf</td>
</tr>
<tr>
<td>Solver options frame</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>&quot;ode1&quot;</td>
</tr>
<tr>
<td>Mode</td>
<td>&quot;SingleTasking&quot;</td>
</tr>
<tr>
<td>Fixed step size</td>
<td>&lt;time value in seconds&gt;</td>
</tr>
</tbody>
</table>

As a result, the Preferences dialog could look as follows:
4 Clear the Block reduction checkbox on the Advanced page, because this option is not supported by RTI. Otherwise, the build procedure will fail.

As a result, the Preferences dialog could look as follows:

Once a new model is saved, the simulation parameters are stored with it. Therefore, changing a default value in the Preferences dialog does not affect your existing models.
How to Specify RTI-Specific Settings for a Model

Suppose you want to migrate a model to a different RTI, for example, from RTI1103 to RTI1104. Or you want to use an existing Simulink model with RTI for the first time. Carry out the following steps.

To specify RTI-specific settings for a model

1. Change to the desired RTI.
2. Open the model you want to convert.
3. Enter `set_rti` in the MATLAB Command Window.

This function configures the simulation parameters of the current model to settings suitable for code generation with the currently active platform, for example, the system target file, template makefile, make command, etc.

You might also need to replace the I/O blocks of the model with I/O blocks for the new platform.
How to Remove the RTI-Specific Settings from a Model

When a model is configured for use with RTI, it contains not only RTI-specific blocks but also block-independent, RTI-specific settings. These remain in the model even if you remove all the RTI-specific blocks. If you open such a model with a non-RTI environment, MATLAB issues appropriate warnings. Therefore, you should use the \texttt{rti.mdlcleanup} function to remove all the RTI-specific settings from the model and avoid the warnings.

\textbf{To remove the RTI-specific settings from a model}

\begin{enumerate}
\item Open the model you want to convert.
\item Delete all RTI blocks from the model.
\item Enter \texttt{rti.mdlcleanup} in the MATLAB Command Window.
\end{enumerate}

This function removes all RTI-specific settings, and closes and reopens the model.
Defining I/O Interfaces

To connect the real-time simulation to the physical world, you need to introduce I/O interfaces into your model. These allow you to replace parts of your simulated model with real hardware. In RTI and RTI-MP, the I/O of each dSPACE board is supported by one or more I/O libraries that contain the corresponding I/O blocks.

If you want to generate an S-function from a model or subsystem using the RTW S-Function, you cannot use any I/O blocks in that model.

**Naming conventions**

All dSPACE I/O blocks follow certain naming conventions. Refer to Naming Conventions on page 41.

**PHS-bus address**

For modular dSPACE systems there is a distinct relationship between the PHS-bus address of an I/O board and its board number as shown in RTI. Refer to PHS-Bus Address and Board Number on page 43.

**Adding I/O blocks**

Adding a dSPACE I/O block to a Simulink model is almost like adding any Simulink block. Refer to How to Add I/O Blocks on page 45.
### Characteristics of I/O blocks

The dSPACE hardware provides vectorized and scalar I/O, each of which has certain advantages. Refer to Vectorized Versus Scalar I/O Blocks on page 47.

The dSPACE I/O is capable of using Simulink’s data typing feature:
- The different I/O blocks use certain data types. Refer to Data Typing Basics on page 49.
- Data typing can be turned on or off. Refer to How to Enable Data Typing on page 50.

### Related Topics

#### Restrictions with data typing

Certain restrictions apply when using Simulink’s data typing feature. Refer to Data Typing Restrictions on page 273.

#### I/O in custom C code

There are a few I/O features that are not available as RTI blocks. You can use these features via S-functions or User-Code, which lets you implement user-defined functions. Refer to Inserting Custom C Code on page 153.

#### Multiprocessor models

If you run a multiprocessor system, you also have to distribute the model to the various CPUs. Refer to Distributing the Model for MP Systems on page 117.
Naming Conventions

All I/O blocks have default names based on dSPACE’s board naming conventions:

- Block names always start with the board name.
- The description of their functionality is added.
- In modular systems, the suffix B and the board number are added. If your system contains several boards of the same type, this board number enables RTI to distinguish between them. The board numbering restarts with 1 for each board type. For details on the relation between the PHS-bus address of an I/O board and the corresponding board number, refer to PHS-Bus Address and Board Number on page 43.
- For MicroAutoBox, the suffix M and the module number are added. If your MicroAutoBox contains several modules of the same type, this module number enables RTI to distinguish between them. The module numbering restarts with 1 for each module type.
- If the I/O unit allows channel-wise access, the suffix C and the selected channel number are added.
- There are further suffixes: G specifies a group number, CON a converter number, I an interrupt number, P a port number, and BL a channel block.

If you customize the name of a block, it does not correspond to dSPACE’s naming conventions, and is therefore displayed in italics. Do not use the “ character (quotation mark) in block names.

⚠️ When you copy an I/O block, Simulink might modify the board, channel, group number, etc. in order to keep the block names unique. This inherently modifies the settings of the copied block.
The following illustration shows a few examples of I/O block names:

- MUX ADC
  - DS1104MUX_ADC

- ADC
  - DS1104ADC_CE

- BIT IN
  - DS2201IN_B1_C1

- MASTER BIT IN
  - Custom Name

- Channel number
- Board number
- No number
- Customized name in italics
PHS-Bus Address and Board Number

If you have a modular system that contains several I/O boards of the same type, you can distinguish them via their PHS-bus addresses, which make them unique in your system (the PHS-bus address of each board was specified during the hardware installation – refer to How to Set the PHS-bus Address in the corresponding Installation and Configuration Guide for details).

ControlDesk's Platform Manager sorts I/O boards of the same type in ascending order of their PHS-bus addresses. The same ascending order applies to the board numbers (the number after the suffix B) of the corresponding I/O blocks in Simulink.

- As the hardware design of DS4201 and DS4201-S boards is very similar, ControlDesk's Platform Manager cannot handle them as different board types; thus, you have to consider their PHS-bus addresses to find the correct board numbers for RTI's I/O blocks.

- To create a model without having the dSPACE system available, follow the guidelines in How to Create an Application for a Virtual dSPACE System on page 265 to find I/O block settings that match the actual hardware.
The following table demonstrates how the Platform Manager lists the different boards of a fictional modular DS1005 or DS1006 system and which board numbers result for RTI's I/O blocks:

<table>
<thead>
<tr>
<th>Board Type</th>
<th>Display in ControlDesk's Platform Manager</th>
<th>PHS-Bus Offset</th>
<th>Board Number (RTI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS2001</td>
<td>DS2001</td>
<td>0x00</td>
<td>1</td>
</tr>
<tr>
<td>DS2001</td>
<td>DS2001</td>
<td>0x20</td>
<td>2</td>
</tr>
<tr>
<td>DS2001</td>
<td>DS2001</td>
<td>0x40</td>
<td>3</td>
</tr>
<tr>
<td>DS5001</td>
<td>DS5001</td>
<td>0x30</td>
<td>1</td>
</tr>
<tr>
<td>DS5001</td>
<td>DS5001</td>
<td>0x50</td>
<td>2</td>
</tr>
<tr>
<td>DS4201</td>
<td>DS4201</td>
<td>0x10</td>
<td>1</td>
</tr>
<tr>
<td>DS4201-S</td>
<td>DS4201</td>
<td>0x60</td>
<td>2</td>
</tr>
<tr>
<td>DS4201-S</td>
<td>DS4201</td>
<td>0x70</td>
<td>3</td>
</tr>
<tr>
<td>DS4201</td>
<td>DS4201</td>
<td>0x80</td>
<td>4</td>
</tr>
</tbody>
</table>

The following table demonstrates how the Platform Manager lists the different modules of a fictional modular DS1401 system and which module numbers result for RTI's I/O blocks:

<table>
<thead>
<tr>
<th>Module Type</th>
<th>Display in ControlDesk's Platform Manager</th>
<th>Module Position</th>
<th>Module Number (RTI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC Type</td>
<td>ADC_TYPE1_M1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CAN Type</td>
<td>CAN_TYPE1_M1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>DIO Type</td>
<td>DIO_TYPE1_M1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>DAC Type</td>
<td>DAC_TYPE1_M1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>ECU Type</td>
<td>ECU_TYPE1_M1</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>
How to Add I/O Blocks

Consider the throttle control example (see Throttle Control Example on page 20) and suppose you want to carry out rapid control prototyping. You have to replace the simulated throttle valve with the real one, which entails exchanging the corresponding Simulink blocks of your model with suitable I/O blocks. To run the real-time simulation of this model you have to connect the corresponding I/O to the real throttle valve. Some further adjustments might also be necessary in the model, for example, adding linearization, normalization and saturation.

If you have a MicroAutoBox 1401/1501, the resulting Simulink model would resemble the one shown in the illustration below (it is a simplified model though).

The blocks used for input and output in this example have different characteristics: The ADC block acting as input provides a single vectorized port. The output is done via a PWM generation block where the output frequency is constant but the duty cycle is varied.
You must not break the library links of the RTI blocks. Otherwise, the blocks cannot be updated when you migrate your model to a new version of RTI and it is very likely that your model will not operate correctly under such a condition.

To add I/O blocks to the model

1. In the Library Browser, expand the dSPACE I/O library that corresponds to the particular I/O unit you want to address.
2. Drag and drop the appropriate block from the I/O library to your Simulink model. You can place RTI's I/O blocks on the root level or in any subsystem.
3. Connect the block to your model as you would with any standard Simulink block and open the Block Parameter dialog.
4. Edit the settings of the block to meet your requirements: for example, the board number, channel number, and range. For details on the different I/O blocks and their settings, refer to the corresponding reference material in the dSPACE HelpDesk.

   Most of RTI's I/O blocks apply automatic scaling to their Simulink ports. An input block provides floating-point values in the range $-1 \ldots +1$. An output block requires values in the same range. You should make maximum use of this $-1 \ldots +1$ range to exploit the full resolution of your converters.

   In order to optimize the performance of your application, RTI does not generate any code for unused channels. (This does not apply to several I/O blocks of RTI1401 where I/O code is not generated but implemented as S-functions.) If you leave unused channels open, Simulink produces a warning message by default.

To avoid warnings due to unconnected ports

- Connect all unused ports to **Ground** or **Terminator** blocks from the Simulink library.
Vectorized Versus Scalar I/O Blocks

RTI provides two kinds of I/O blocks: vectorized and scalar. Both types have specific characteristics and require different handling. The following table lists the main differences:

<table>
<thead>
<tr>
<th></th>
<th>Scalar I/O Blocks</th>
<th>Vectorized I/O Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access of I/O device</td>
<td>Individual (scalar) I/O channels</td>
<td>Groups of (vectorized) I/O channels</td>
</tr>
<tr>
<td>Type of ports</td>
<td>One or more scalar ports</td>
<td>One or more vectorized ports</td>
</tr>
<tr>
<td>Multiple blocks</td>
<td>A single I/O device may be accessed via several instances of the same I/O block</td>
<td>A single I/O device can be accessed by only a single I/O block</td>
</tr>
<tr>
<td>Sample times</td>
<td>Individual channels may have different sample times</td>
<td>The elements of a vector always inherit the same sample time</td>
</tr>
</tbody>
</table>

For details on the different I/O blocks and their settings, refer to the corresponding reference material in the dSPACE HelpDesk.

**Scalar I/O blocks**

- **Channel numbering** The numbering of the different I/O channels is top down with respect to the I/O block.

- **Multiple blocks** This feature of scalar I/O blocks can help you avoid long connection lines because you can place several copies of the same block wherever you need them in your model. However, you have to ensure that you do not access any channel more than once.

**When you copy an I/O block, Simulink might modify the board, channel, group number, etc. in order to keep the block names unique. This inherently modifies the settings of the copied block.**
Vectorized I/O blocks

I/O access In the dialog of the I/O block you can specify which vector elements (channels) you want to activate. The number of enabled elements defines the vector width of the I/O block’s port(s).

Sample times If you connect a vectorized I/O block to Simulink blocks that have different sample times, the vectorized I/O block inherits the fastest sample time. This might lead to illegal sample-rate transitions in some cases. If you get such an error message while compiling the model, you have to follow the instructions provided in *How to Achieve Data Consistency Between Timer Tasks* on page 70.
Data Typing Basics

MATLAB and Simulink support the following data types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>Double-precision floating point</td>
</tr>
<tr>
<td>single</td>
<td>Single-precision floating point</td>
</tr>
<tr>
<td>int8</td>
<td>Signed 8-bit integer</td>
</tr>
<tr>
<td>uint8</td>
<td>Unsigned 8-bit integer</td>
</tr>
<tr>
<td>int16</td>
<td>Signed 16-bit integer</td>
</tr>
<tr>
<td>uint16</td>
<td>Unsigned 16-bit integer</td>
</tr>
<tr>
<td>int32</td>
<td>Signed 32-bit integer</td>
</tr>
<tr>
<td>uint32</td>
<td>Unsigned 32-bit integer</td>
</tr>
<tr>
<td>Boolean</td>
<td>(0 or 1) represented internally by uint8 values</td>
</tr>
</tbody>
</table>

RTI uses the following data types:
- Single-bit I/O blocks use the Boolean data type. Therefore, you can connect them directly to all logical operators.
- Byte I/O blocks use the uint8 data type.
- Encoder Index Search blocks use the int8 data type.
- DAC and ADC blocks use the double data type.

There are a few restrictions concerning the use of data typing (see Data Typing Restrictions on page 273).
How to Enable Data Typing

By default, this feature is disabled to grant downward compatibility. In this case the default data type (double) is used. However, if you want to enable data typing, you have to carry out the following instructions.

To enable data typing

➤ Open the Advanced page of the Simulation Parameters dialog, and set the **Boolean logic signals** to **on** as shown in the illustration below.

Although this option reads “Boolean logic signals”, with RTI and RTI-MP, it controls the overall use of data typing, not only that for the Boolean data type.

RTI blocks that were introduced after The MathWorks added the data typing feature always use their specific data type, regardless of whether data typing is enabled or not. Downwards compatibility is not an issue for these newer RTI blocks.

For further information on data typing, refer to *Using Simulink* by The MathWorks.
Handling Tasks

Your Simulink model usually consists of parts that have to be calculated periodically and others that have to be calculated at aperiodic points in time. Therefore, the real-time code is not executed as a whole, but Real-Time Workshop and RTI subdivide it according to the model structure and assign the parts to tasks. The tasks are scheduled individually so that they execute when necessary, without a fixed execution order. The request for the execution of a particular task is called an interrupt and can come, for example, from a timer device of the processor or from an I/O device. Interrupts can be internal or external events, and they can be periodic or aperiodic. Each model has at least one timer task (executed with the base sample time) or a task driven by an interrupt block.
Consider the throttle control example (see Throttle Control Example on page 20). Suppose you want to monitor the flow rate of the throttle valve with a long sample time. To complete this job you need to understand how timer tasks function so that you can specify the desired sample time and select a suitable timer task mode. Suppose you also want to include an emergency stop button that immediately closes the throttle valve when it is pressed. For this job you have to implement a task driven by a hardware interrupt block.

**Basics**

RTI uses the dSPACE Real-Time Kernel to schedule the various tasks. Refer to Real-Time Kernel and Scheduler on page 55.

Each task needs to have a priority. Refer to Priorities and Task-Switching Time on page 56.

The scheduler assigns a task state to each task, which identifies whether the task is running, ready or idle. Refer to Task States and Execution Order on page 58.

If a task is requested to start even though it is still running, an overrun situation can occur. Refer to Overrun Situation and Turnaround Time on page 60.

**Timer tasks**

Periodic execution of a timer task, for example, the sampling of an input at equidistant intervals. If such a job is not controlled by a designated timer, equidistant intervals cannot be achieved. Refer to Timer Tasks on page 63.

**Tasks driven by interrupt blocks**

Aperiodic or periodic execution of a task driven by an interrupt block. An example of an aperiodic task is an input signal indicating an erroneous situation. This typically requires instantaneous treatment. Refer to Tasks Driven by Interrupt Blocks on page 75.

**Background tasks**

The background task is executed whenever no other task is active. Because it is interrupted by all other tasks, it is not suitable for real-time jobs. The calculations made in it do not have specific timing, but are either distributed over several base sampling steps or executed several times between two base sampling steps. This is why it mainly deals with administrative jobs. Refer to Assigning Parts of the Model to the Background Task on page 93.
Even though it is not visible in the Simulink model, there is also a background task in the throttle control example: The mechanism that starts or stops the simulation is located there, for example.

**Configuring priorities**
The priorities of the tasks specify their execution order in case of mutual interruptions. Refer to *How to Change the Task Priorities* on page 96.

**Configuring overrun strategies**
The overrun strategy for a task specifies how RTI behaves if an overrun occurs for the task. Refer to *How to Handle Overrun Situations* on page 99.

**Combining interrupts**
Sometimes several interrupt sources have to be combined to trigger a single subsystem. Refer to *How to Combine Several Interrupt Sources* on page 101.

**Subscheduling tasks**
Sometimes an interrupt-block-driven task has to be divided into several subtasks that are scheduled individually. Refer to *How to Subschedule Tasks* on page 104.

**Time-triggered tasks and timetable tasks**
In contrast to event-based execution of tasks, this blockset lets you specify time-triggered task execution. Time-triggered tasks and timetables let you implement task trigger times and even complete time-dependent schedules (i.e., timetables) for tasks. You should read up on the basics of the two techniques. Refer to *Basics of Time-Triggered Tasks and Timetables* on page 106.

Time-triggered tasks let you start a task with one or more trigger delay times. The trigger delay time is a Simulink signal. Refer to *How to Implement Time-Triggered Tasks* on page 110.

A timetable lets you start a set of timetable tasks in relation to a single trigger event. The start delay time for the timetable is a Simulink signal. Refer to *How to Implement Timetables* on page 113.
**Related Topics**

**Task information variables**

RTI and RTI-MP generate certain task information variables for each task of the real-time application. These variables are available in the variable description file and allow you to collect some important information about the real-time application, for example, how much execution time a task needs, what priority and state it has, and whether it ran into an overrun situation. For further information, see *RTI and RTI-MP Variable Reference* in the *RTI and RTI-MP Implementation Reference*. 
Real-Time Kernel and Scheduler

RTI uses dSPACE Real-Time Kernel (RTK) to manage tasks, handle interrupts and schedule an application's tasks efficiently. The RTK provides a priority-based preemptive scheduler that supports the rate-monotonic scheduling (RMS) strategy. By means of the priority of a task, the scheduler decides whether it should be started immediately or if it has to wait until a higher-priority task is completed. Hence, higher-priority tasks interrupt lower-priority tasks. Interrupt handling and scheduling cause a small time overhead.
Priorities and Task-Switching Time

You might want to assign a priority to each task of your application.

Consider the following situation in the throttle control example: Someone presses the emergency stop button. Thus, the corresponding task of the control unit that closes the throttle valve must have the highest priority. All other tasks of the control unit have to wait. When the valve is closed, the suspended tasks resume execution.

Therefore, if the execution request for the high-priority task occurs, the scheduler of the control unit has to carry out the following steps:

- Interrupt (preempt) the currently running (lower-priority) task;
- Execute the requested high-priority task;
- When the high-priority task is finished, resume the interrupted (lower-priority) task.

The priorities of the tasks are stored in variables of the real-time application and can be displayed in ControlDesk (see priority in the RTI and RTI-MP Implementation Reference).

The background task is not a task in the common sense, meaning that it has no priority.

A crucial value in task scheduling is the task-switching time, which defines the delay between the occurrence of an interrupt on the hardware and the execution of the corresponding task’s first statement. In the example mentioned above, this is the time that passes until the control unit responds to a person pressing the emergency stop button.
In a multiple-task environment, a triggered task with the highest priority has the following task-switching times:

<table>
<thead>
<tr>
<th>Board</th>
<th>Best Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1005 @ 480 MHz</td>
<td>0.95 µs</td>
<td>1.9 µs</td>
</tr>
<tr>
<td>DS1005 @ 800 MHz</td>
<td>0.71 µs</td>
<td>1.42 µs</td>
</tr>
<tr>
<td>DS1005 @ 1.0 GHz</td>
<td>0.66 µs</td>
<td>1.32 µs</td>
</tr>
<tr>
<td>DS1006 @ 2.2 GHz</td>
<td>4.15 µs</td>
<td>8.3 µs</td>
</tr>
<tr>
<td>DS1103 @ 333 MHz</td>
<td>1.62 µs</td>
<td>3.24 µs</td>
</tr>
<tr>
<td>DS1103 @ 400 MHz</td>
<td>1.03 µs</td>
<td>2.06 µs</td>
</tr>
<tr>
<td>DS1103 @ 1.0 GHz</td>
<td>0.35 µs</td>
<td>0.7 µs</td>
</tr>
<tr>
<td>DS1104 @ 250 MHz</td>
<td>2.4 µs</td>
<td>4.8 µs</td>
</tr>
<tr>
<td>DS1401 @ 200 MHz</td>
<td>1.44 µs</td>
<td>2.9 µs</td>
</tr>
<tr>
<td>DS1401 @ 300 MHz</td>
<td>1.19 µs</td>
<td>2.38 µs</td>
</tr>
<tr>
<td>DS1401 @ 800 MHz</td>
<td>0.5 µs</td>
<td>1.0 µs</td>
</tr>
</tbody>
</table>

The stated task-switching times apply only if the involved routines are already located in the cache. Otherwise, the task-switching times can be significantly longer.
Task States and Execution Order

In order to identify which task is active, which one is suspended, and so on, the scheduler assigns a state to each task as shown in the following illustration:

The dSPACE Real-Time Kernel uses the definitions listed in the table below:

<table>
<thead>
<tr>
<th>State</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>0</td>
<td>The task is inactive, waiting to be triggered.</td>
</tr>
<tr>
<td>Ready</td>
<td>1</td>
<td>The task has been triggered but could not start due to a high-priority task that is currently running. It is waiting to start the execution.</td>
</tr>
<tr>
<td>Running</td>
<td>2</td>
<td>The task has started running. This state is true until the task finishes running, regardless of whether it is suspended by a high-priority task or not.</td>
</tr>
</tbody>
</table>

Depending on the priorities and current states of the tasks, the scheduler executes them according to the following rules:

- A high-priority task that is triggered always suspends a low-priority task that is currently running.
- If no high-priority task is triggered, the suspended low-priority task resumes execution.
- Tasks of the same priority do not suspend each other if they are triggered, but follow a first come, first served policy.
- As long as all other tasks are idle, the background task is executed.
The states of the tasks are stored in variables of the real-time application and can be displayed in ControlDesk (see state in the RTI and RTI-MP Implementation Reference).
Overrun Situation and Turnaround Time

An overrun situation occurs if a task is requested to start but has not finished its previous execution yet. Consider the situation shown below:

Between the first and the second timer interrupt there is no overrun, but if the hardware-interrupt-block-driven task $T_{HW}$ needs to be calculated, there is not enough time for the timer task $T_{timer}$ to finish until it is requested to start again. If such an overrun situation occurs for a task, the scheduler detects it, and RTI responds according to the overrun settings for the task (see How to Handle Overrun Situations on page 99).

To avoid overrun situations you have to consider the time values listed below. For each task of your model, ensure that its sample time is greater than the sum of these time values.

- Depending on the dSPACE board used, you have to take into account a different task-switching time as stated in the above-mentioned table.

The turnaround time for a task is the time that passes between the triggering and the end of its execution. It can include the time required by higher-priority tasks that interrupt it.
Conditional execution paths in the model code can cause the turnaround time of a task to vary from one execution step to another. For example, if you run your model in the multiple timer task mode, the fastest task actually contains only blocks that are executed with this sample time. Nevertheless the turnaround time of this task might have periodical peaks occurring with the period of a slower task, which result from the interaction with the slower timer tasks.

See also turnaroundTime in the RTI and RTI-MP Implementation Reference. In the real-time application, you can measure the turnaround time of each task via ControlDesk. To get a continuous display of the turnaround time, you must use a plotter instrument. For information on connecting a variable to a ControlDesk instrument, refer to Creating Data Connections in the ControlDesk Experiment Guide.
The illustration below shows the turnaround times for a number of tasks.

You can configure the behavior towards overrun situations for each task individually, see How to Handle Overrun Situations on page 99.


Timer Tasks

RTI provides two execution modes to handle the different sample times of a model.

**Basics**

You should know how a model part becomes part of a timer task and how to distinguish single-rate from multirate models. Refer to Timer Task Basics on page 64.

**Execution modes**

The single timer task mode combines all sample times in one single timer task. Refer to Single Timer Task Mode on page 65.

The multiple timer task mode forms an individual timer task for each sample time. Refer to Multiple Timer Task Mode on page 66.

Each execution mode has specific advantages and disadvantages, which can help you choose the correct execution mode. Refer to Comparing the Execution Modes on page 73.

**Synchronizing timer tasks to external signals**

You can bind the timer tasks of your model explicitly to a hardware interrupt block; for example, to synchronize the model evaluation to an external timing signal. Refer to How to Bind a Timer Task Explicitly to an Interrupt Block on page 73.

**Related Topics**

RTI models without timer task

With single-processor systems it is possible to create a model driven only by interrupt blocks by placing all the non-virtual blocks in or below function-call subsystems. However, for such a model the currentTime variable remains zero throughout the simulation because Real-Time Workshop calculates the currentTime variable in the fastest timer task and that task does not exist for models driven only by interrupt blocks. As a consequence, blocks that depend on the absolute simulation time do not work (see General Limitations on page 275).
Timer Task Basics

If a model part is not driven by an interrupt block, it is part of a timer task by default. The model code for timer tasks is bound implicitly to the timer interrupt source of the processor.

A few Simulink blocks allow you to explicitly specify a sample time or switch to sample-based operation: for example, the Sine Wave block. If you specify a sample time or switch to the sample-based operation, this block also changes its symbol to indicate that it has become a discrete block. All other blocks – including all RTI blocks – inherit their sample times from the blocks they are connected to. A block that is a signal source inherits its sample time from the blocks connected to its output ports; other blocks inherit their sample times from the blocks connected to their input ports.

Single-rate and multirate models

In general, one or more blocks in the model require periodic execution at the base sampling rate. All these blocks are grouped in the first timer task, which is bound to a timer interrupt. If the model also contains at least one time-discrete block with a sampling rate different from the base sampling rate, the model is multirate (otherwise it is single-rate). The different sampling rates of a multirate model can be calculated in one single timer task or in several timer tasks – one for each sampling rate. With Real-Time Workshop, the following formula applies to all sampling rates:

\[
\text{sampling rate} = \frac{1}{n} \cdot \text{base sampling rate}
\]

where

\[
\begin{align*}
    n &= \text{integer multiple} \\
    \text{sampling rate} &= \frac{1}{\text{sampling time parameter}} \\
    \text{base sampling rate} &= \frac{1}{\text{fixed step size of model}}
\end{align*}
\]
Consider the throttle control example (see Throttle Control Example on page 20). To measure the flow rate of the throttle valve at a slow sample time, you have to alter the model as shown in the next illustration. The upper part operates with the base sample time. The lower part that monitors the flow rate operates at a slow sample time of just one sample per second.

**Single Timer Task Mode**

The single timer task mode is the default execution mode. RTI places all calculations performed in timer tasks in one single timer task. This is executed with the fastest sample time of your model, that is, the base sample time.

If your model is a multirate system, a sample hit control mechanism ensures that the discrete blocks are executed at their specified sampling steps. The example below demonstrates this case.
Two discrete blocks ($B_{\text{Fast}}$ and $B_{\text{Slow}}$) are executed in a single timer task $T_{\text{Timer}}$ with the block $B_{\text{Slow}}$ being executed only every second step:

There is no preemption mechanism for the timer tasks of the model in this execution mode. Therefore, the base sample time must be long enough to allow the execution of all blocks within one interval. If this is not possible because you require a smaller base sample time, you get an overrun situation. This is illustrated in the next example.

If the base sample time is too small, an overrun situation occurs:

You can configure the behavior towards overrun situations for each task individually, see How to Handle Overrun Situations on page 99.

**Multiple Timer Task Mode**

The multiple timer task mode is the optional execution mode in which RTI creates several timer tasks, one for each sample time/sample time offset of your model. Therefore, the execution of a slow block can be interrupted (preempted) for the execution of a fast (higher-priority) block. This mechanism can solve the overrun problem shown in Single Timer Task Mode on page 65.
Basics
The multiple timer task mode creates one timer task per sample time and sample time offset. Refer to Basics of the Multiple Timer Task Mode on page 67.

Example
Certain types of applications require the multiple timer task mode. Refer to Example Applications Requiring the Multiple Timer Task Mode on page 69.

Achieving data consistency
Data transfer between different timer tasks needs to be prepared, otherwise inconsistent data can result. Refer to How to Achieve Data Consistency Between Timer Tasks on page 70.

Basics of the Multiple Timer Task Mode
RTI creates several timer tasks, one for each sample time/sample time offset of your model. Therefore, the execution of a slow block can be interrupted (preempted) for the execution of a fast (higher-priority) block.

Sample time offset
If you use sample time offsets, each offset is treated as a new task: The smaller the sample time offset, the higher the priority of a task. Refer to Using Simulink by The MathWorks.

If block BFast is assigned to task TTimer1 and block BSlow to task TTimer2, the execution sequence in the multiple timer task mode runs as shown below:

After TTimer1 has finished, TTimer2 starts. The next time TTimer1 is ready for execution, TTimer2 is interrupted until task TTimer1 has finished. TTimer2 then resumes its execution.
As long as all tasks can be executed within the time interval given by the base sample time, the model behaves in a manner similar to the execution in the single timer task mode.

In the multiple timer task mode, RTI manages the priorities of the different timer tasks according to a rate-monotonic scheme: The faster the sample time of a task, the higher its priority.

For a demo of multiple timer tasks, see the Multiple Timer Tasks demo model, which you can access via the RTI block library – Demos – TaskLib Demos.

Because there are no true aperiodic interrupts in a Simulink simulation, Simulink and Real-Time Workshop use the terms SingleTasking and MultiTasking instead of single timer task mode and multiple timer task mode. However, with dSPACE real-time programs you can place all calculations performed in timer tasks in a single timer task and still have additional tasks triggered by hardware or software interrupt blocks. Therefore, RTI uses the terms single timer task mode and multiple timer task mode – which both allow aperiodic interrupts.
Example Applications Requiring the Multiple Timer Task Mode

The following example applications require the multiple timer task mode.

**FFT**

The multiple timer task mode is required if you want to use the DSP Blockset to run an FFT in addition to regular continuous or discrete blocks. The FFT is not started until the required data (for example, 1024 points) is available, and it does not finish in just one interval, as illustrated below:

**Aperiodic interrupt**

Another example where the multiple timer task mode can avoid overrun situations is illustrated below. Here you can see an aperiodic interrupt in a multirate application that delays the execution of the timer task so that it cannot finish before the next timer interrupt occurs:

![Diagram showing an example application requiring the multiple timer task mode.](image-url)
If you choose the multiple timer task mode for this model, the fast block $B_{Fast}$ can interrupt the slow block $B_{Slow}$ so that no task is required to start executing while it is still running. Therefore, there is no overrun situation, which is illustrated below:

**How to Achieve Data Consistency Between Timer Tasks**

In the multiple timer task mode, each sample time forms a separate timer task and these timer tasks can interrupt each other. If data vectors were transferred directly between blocks with different sample times (called sample-rate transition), data inconsistency could result.
Suppose a timer task with a fast sample time ($T_{Fast}$) provides the data for a timer task with a slow sample time ($T_{Slow}$). $T_{Fast}$ might be executed while $T_{Slow}$ is busy reading the data vector – marked with (1) in the following illustration. Thus, $T_{Slow}$ is interrupted and partly reads old data (2) and new data (3) when it resumes its execution. A similar problem arises if a slow timer task provides data for a fast one. Both of these cases of data inconsistency are illustrated below:

Because of this data inconsistency you cannot connect blocks that are executed with different sample times directly. Instead, you must insert the Rate Transition block from the Simulink library. This block is executed with the priority of the faster block, but with the sample time of the slower block.

- If you select the “Fast to slow” Data rate transition, the block is executed after the fast block and before the slow block – marked with a (1) in the following illustration – thereby providing consistent data for the slow block (2).

- If you select the “Slow to fast” Data rate transition, the block is executed after the slow block and before the fast block (3), thereby providing consistent data for the fast block (4).
The Rate Transition block is available since MATLAB R13. However, rate transitions via Zero Order Hold or Unit Delay blocks are still valid, so you do not need to modify your old models.

The illustration below demonstrates the data transfer for both “fast to slow” (f2s) and “slow to fast” (s2f) data transitions:

![Data Transfer Illustration](image)

The following instructions show you how to implement the sample-rate transitions.

**To implement sample-rate transitions**

1. Introduce the Rate Transition block from the Simulink library into the connection of the two blocks.

2. Open the Block Parameters dialog and select the correct **Data rate transition**.

   Inserting a “Slow to fast” data rate transition changes the model dynamics because it delays the output of the slower block by one time step.
Via the **Format – Sample time colors** menu command, you can make Simulink display blocks and signal lines in different colors for the different sample times. As a result, you can easily see where two adjacent blocks are different colors, indicating a sampling rate transition. You should add the Rate Transition block at this position.

### Comparing the Execution Modes

The following table gives an overview of the advantages and disadvantages of both execution modes:

<table>
<thead>
<tr>
<th></th>
<th>Single Timer Task Mode</th>
<th>Multiple Timer Task Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Time Overhead</td>
<td>Smaller</td>
<td>Bigger</td>
</tr>
<tr>
<td>Target Application</td>
<td>• Single-rate</td>
<td>• Multirate with a predominating timer task that has a short sample time</td>
</tr>
<tr>
<td></td>
<td>• Multirate with a predominating timer task that has a long sample time</td>
<td>• Multirate with a predominating task driven by an interrupt block</td>
</tr>
<tr>
<td>Conclusion</td>
<td>Standard applications</td>
<td>Solving specific overrun problems</td>
</tr>
</tbody>
</table>

### How to Bind a Timer Task Explicitly to an Interrupt Block

By default, the timer tasks of a model are connected to a timer interrupt of the dSPACE hardware. As a result the model is executed at fixed time steps. However, sometimes you might want to compute the model synchronously to external events, for example, an external timer device. Therefore, you have to connect the timer task(s) to an external interrupt source. This is possible via the Timer Task Assignment block.
The computations of the timer task are executed as normal using the time value of the simulation step size, as specified in the Simulation Parameters dialog or Multiprocessor Setup dialog. However, the points in time when the computations are performed depend on the external trigger events. RTI cannot check whether the simulation step size of the model and the step size of the external trigger are equal. You must ensure identical step sizes yourself. Otherwise, the model is not calculated in real time.

To connect the timer task(s) of a model to an external interrupt

1. Place the desired hardware interrupt block (see example on the left) from the appropriate I/O library on the root level of your Simulink model.
2. Open the Block Parameters dialog, and specify the settings you require, for example, the interrupt type. If you use a modular system, you also have to specify the correct board number.
3. Place the Timer Task Assignment block (see on the left) from RTI's TaskLib on the root level of your model.
4. Connect the output port of the hardware interrupt block to the input port of the Timer Task Assignment block.
5. Open the Simulation Parameters dialog or Multiprocessor Setup dialog, and make sure that the models Fixed Step Size or Basic Step Size and Step Size Multiple match the period of the external interrupt source.

For a demo of this task type, see the Timer Task Assignment demo model, which you can access via the RTI block library – Demos – TaskLib Demos.
Tasks Driven by Interrupt Blocks

RTI and RTI-MP let you drive tasks by several interrupt blocks to achieve flexible task handling.

**Triggered subsystems**
Simulink provides several types of triggered subsystems that serve different needs, for example, for driving a model part by an interrupt block. Refer to Types of Triggered Subsystems on page 76.

**Preparing a subsystem**
For a subsystem to be driven by an interrupt block it has to be prepared. Refer to Preparing a Function-Call Subsystem on page 78.

**Hardware interrupts, timer interrupts**
Hardware and timer interrupts are generated by designated hardware or timer interrupt devices of the dSPACE boards, for example, if a specific input signal occurs. Refer to How to Use Hardware Interrupts on page 87.

**Software interrupts**
Software interrupts are generated by the model itself, for example, if a signal exceeds a specific value. Refer to How to Use Software Interrupts on page 89.

**Example**
For an example of a task driven by an interrupt block including the necessary task transitions, see Example of a Task Driven by a Hardware Interrupt Block on page 91.

**Related Topics**

**Code reuse feature**
Real-Time Workshop is able to generate subsystems as functions that can be reused. However, with RTI certain limitations apply to reused subsystems. For details, refer to How to Bundle the Execution of Blocks on page 214.
Types of Triggered Subsystems

To drive a part of the model by a hardware or software interrupt block, the part has to be placed in a specially triggered subsystem. Simulink has several types of subsystems:

**Subsystem without Trigger or Enable block**  This simply helps to structure your model layout; for instance, to group parts that logically belong together, such as a controller. If declared atomic, a subsystem also influences the block execution order and thus the generated real-time code (see Optimizing the Build Process and Simulation via Atomic Subsystems on page 213).

If you add an Enable or Trigger block to a subsystem it becomes a conditionally executed subsystem as shown in the following paragraphs.

**Subsystem with a Simulink Enable block**  This executes whenever the enable signal is greater than zero. This type of conditionally executed subsystem does not form a separate task in the real-time program but is part of its parent subsystem's task. The enable port of the subsystem requires a Simulink signal and cannot be used with hardware or software interrupts.

**Subsystem with a Simulink Trigger block**  executes once whenever an appropriate trigger signal is applied to the trigger port of the subsystem.

- If the Trigger type is set to “rising,” “falling,” or “either”, the trigger port of the subsystem requires a Simulink signal and cannot be used with hardware or software interrupts. This type of conditionally executed subsystem does not form a separate task in the real-time program but is part of its parent subsystem's task.

- If the Trigger type is set to “function-call”, the trigger port of the subsystem accepts only function-call connections (for example, from RTI's Interrupt blocks, or the Simulink Function-Call Generator block) and no standard Simulink signals. Throughout this RTI and RTI-MP Implementation Guide, this is referred to as a function-call subsystem. If the trigger port is connected to a Timer, Hardware, or Software
Interrupt block, the function-call subsystem forms a separate task in the real-time program. Therefore, a typical application of this trigger type is for building tasks driven by interrupt blocks. If the trigger port is connected to RTI's Background block, the subsystem is assigned to the background task.

Since MATLAB R13SP1 the Trigger block provides the following options:

- **States when enabling**
- **Sample time type** and **Sample time**

These options have no impact on function-call subsystems that are driven by RTI interrupt blocks. In detail this means that:

- The **States when enabling** option has no effect. Independently of its setting ("inherit", "held", or "reset") the states of a function-call subsystem triggered by an RTI interrupt block will never be reset when an interrupt occurs. This corresponds to the "held" behavior and is the same as in previous releases.

- The **Sample time type** option must always be set to "triggered" (the **Sample time** option is then disabled). Otherwise, you get the following error message from Simulink:

> Block 'test/Function-Call Subsystem1/function' has a discrete sample time. Only constant blocks or blocks with an inherited (-1) sample time are allowed in triggered subsystems.

You can also use the function-call subsystem from the Simulink library or from RTI's TaskLib.
Preparing a Function-Call Subsystem

If you want to drive a particular group of blocks by an interrupt block or assign it to the background task, the blocks must be placed in a function-call subsystem. Data transfer to/from other tasks must be implemented via particular task transitions.

Basics
You should know which task transitions are available for data transfer between interrupt-block-driven tasks and what their characteristics are. Refer to Introduction to Interrupt-Block-Driven Tasks and Task Transitions on page 78.

Preparing a subsystem
The model part you want to drive by an interrupt block must be placed in a function-call subsystem. Refer to How to Prepare a Function-Call Subsystem on page 81.

Implementing task transitions
All data transfer to other tasks must be implemented via valid task transitions. Refer to How to Implement Valid Task Transitions on page 82.

Downward compatibility
Sometimes, you might want to use an old model without modifying its task transitions. Refer to How to Enable Downward Compatibility for Task Transitions on page 86.

Introduction to Interrupt-Block-Driven Tasks and Task Transitions

If you want to drive a particular group of blocks by an interrupt block or assign it to the background task, the blocks must be placed inside a function-call subsystem and the subsystem must be connected to the interrupt source. A function-call subsystem must not be located inside another function-call subsystem.

Since separate tasks can interrupt each other, the transfer of data vectors between them can also be interrupted, producing inconsistent data unless the data transfer is buffered.

With Simulink you can choose whether the transfer of data between different tasks is buffered to ensure data consistency or non-buffered to transfer the data as early as possible but at the cost of potentially inconsistent data vectors.
Non-buffered data transfer

Consider the scenario in the following illustration:

The high-priority task ($T_{\text{High}}$) provides data for the low-priority task ($T_{\text{Low}}$). If $T_{\text{High}}$ is triggered (1) while $T_{\text{Low}}$ is busy reading a part of $T_{\text{High}}$’s last data vector (2), $T_{\text{Low}}$ is interrupted and partly reads old data – it reads new data when it resumes its execution (3).

Buffered data transfer

However, you might require a consistent data transfer as shown in the next illustration:

If $T_{\text{High}}$ is triggered (1) while $T_{\text{Low}}$ reads the old data (3) from buffer A, it writes the new data in buffer B (2). When $T_{\text{Low}}$ resumes execution after $T_{\text{High}}$ has finished (5), it continues reading the old data from buffer A (4). The new data of $T_{\text{High}}$ is not included in this execution of $T_{\text{Low}}$, but the next time $T_{\text{Low}}$ is executed, it reads the new data from buffer B (6).
Valid task transitions

You have to explicitly declare the type of each task transition, i.e., either buffered or non-buffered. This can be done by means of the Non-Buffered Task Transition block for non-buffered data transfer and the Buffered Task Transition (Write) and Buffered Task Transition (Read) blocks for buffered data transfer (see How to Prepare a Function-Call Subsystem on page 81).

Simulink no longer accepts a direct data connection without a task transition block. The Uninterruptable Buffer block is also no longer supported.

Because you need to declare the type (buffered or non-buffered) for each task transition explicitly, Simulink can detect the task transitions that you might have unintentionally left undeclared. Therefore, you do not get an unsafe or problematic model design by mistake.

If your model contains invalid task transitions, for example, a direct data connection, Simulink issues an error message when you attempt to simulate or build the model. However, the error messages might be misleading with regard to the blocks to be used for the task transition (for example, they might recommend using Simulink’s Double Buffer or Rate Transition blocks, which are not compatible with RTI). Make sure that you use the Task Transition blocks provided by RTI’s TaskLib. Simulink’s error messages and the RTI-specific solutions are listed in Checking the Model on page 291.

Downward compatibility

In certain cases, you might want to continue using the previous types of task transitions, meaning direct connection for non-buffered data transfer and connection via the Uninterruptable Buffer block for buffered data transfer. This is possible if you switch RTI to the downward-compatible relaxed task transition check. In this mode, you can use both the old and the new task transition types. Refer to How to Enable Downward Compatibility for Task Transitions on page 86.

The downward-compatible relaxed task transition check will be removed in future versions of RTI, so that you should start to migrate all your models to the new task transition blocks as soon as possible.
How to Prepare a Function-Call Subsystem

All the blocks that are to be driven by an interrupt block or the Background block must be located in a function-call subsystem.

To prepare a function-call subsystem

1. Place the function-call subsystem from RTI's TaskLib inside your Simulink model.

A function-call subsystem must not be located inside another function-call subsystem.

2. Open the function-call subsystem, and add the blocks you want to drive with the interrupt or assign to the background task.

The following illustration shows what your model could look like:
How to Implement Valid Task Transitions

All the data to be transferred between different interrupt-block-driven subsystems or an interrupt-block-driven subsystem and a timer task must be sent via a valid task transition.

- The major benefit of a non-buffered task transition is that data is received as soon as possible, but at the cost of possible data inconsistency. Another benefit is that it does not consume additional memory. (See To implement a non-buffered task transition below.)

- The major benefit of a buffered task transition is that the transferred data read by a task is always consistent, even if a higher-priority task writing new data is triggered meanwhile (see To implement a buffered task transition on page 84).

Each task transition can have receivers in only one task. Therefore, if you need multiple receivers in different tasks, the signal must be branched first and a task transition must be implemented for each branch (see To connect multiple receiving blocks via task transition blocks on page 85).

- The transfer of data between an interrupt-block-driven task and another task (including timer tasks) is not allowed without the task transition blocks from RTI’s TaskLib.

- Task transition blocks support all data types except for complex ones.

- Task transition blocks do not support matrix signals. To transfer a matrix, use the Reshape block from the Simulink library, which converts the signal into a single vector. In the receiving task, you can reshape it back into a matrix.

- Task transition blocks do not support frame-based signals and there is no workaround available.

- Task transition blocks inherit their data types from their sources.

- After simulation start, if the receiving task is executed before the sending task, the output value of the task transition block is not defined. To specify an initial value to be used in this case, you can use Simulink’s IC (Initial Condition) block.
The task transitions described here are not valid for data transfer between two timer tasks. Refer to *How to Achieve Data Consistency Between Timer Tasks* on page 70.

For a demo containing the task transition blocks, see the Task Transition demo model, which you can access via the RTI block library – Demos – TaskLib Demos.

**To implement a non-buffered task transition**

1. Place a Non-Buffered Task Transition block from RTI’s TaskLib directly before or after the function-call subsystem driven by the interrupt block or Background block.

2. To specify an initial value, which the receiver reads until buffered data is available, insert an IC block from the Simulink library into the connection directly after the Non-Buffered Task Transition block.

If the sending and the receiving tasks do not have the same priority and the lower-priority task is interrupted by the higher-priority task, a data inconsistency can occur.
The following illustration shows how you can implement a non-buffered task transition:

```
To implement a buffered task transition

1 Place a Buffered Task Transition (Write) block from RTI’s TaskLib in your model:
   - The signal source is a function-call subsystem: Place the block in the subsystem and connect it directly to the Outport block.
   - The signal source is a timer task: Connect the block directly to the input port of the receiving function-call subsystem.

2 Place a Buffered Task Transition (Read) block from RTI’s TaskLib in your model:
   - If the signal destination is a function-call subsystem, place the block in the subsystem and connect it directly to the Inport block.
   - If the signal destination is a timer task, connect the block directly to the output port of the sending function-call subsystem.
```
3 To specify an initial value, which the receiver reads until buffered data is available, insert an IC block from the Simulink library into the connection directly after the Buffered Task Transition (Read) block.

The following illustration shows how you can implement a buffered task transition:

To connect multiple receiving blocks via task transition blocks
1 Branch the signal inside the sending task for as many receivers as you need.
2 For each branch, implement a buffered or non-buffered task transition as described above.
The following illustration shows how you can branch a signal and send it to multiple receivers. The top branch implements a non-buffered data transfer whereas the bottom branch is buffered. The IC block is left out to simplify the illustration.

How to Enable Downward Compatibility for Task Transitions

If you are reusing a model that was created with RTI 4.2 or earlier and the model contains data connections from or to interrupt-block-driven tasks or the background task, these data connections must be updated to the new task transition blocks. If you cannot modify the model, for example, because you are not its owner, you can switch RTI to the relaxed task transition check. However, you should migrate to the new task transitions as soon as possible, because the relaxed task transition check will be removed in future versions of RTI. The major benefits are:

- Less risk of implementing non-buffered task transitions by mistake.
- Shorter delays when starting tasks because, unlike the Uninterruptable Buffer block, the new Buffered Task Transition blocks disable interrupts only for a very short moment.
To enable the downward-compatible task transition check
➤ Change to the MATLAB Command Window and enter

rti_option TaskTransitionCheck Relaxed

In this mode, you can use direct connections and the old Uninterruptable Buffer block. The new task transition blocks cannot be used.

- The downward-compatible relaxed task transition check will be removed in future versions of RTI, so you should migrate all your models to the new task transition blocks as soon as possible.
- In contrast to the new task transition blocks, no interrupt request is processed while the Uninterruptable Buffer block is busy.
- If the task transition check is relaxed, the check for valid usage of the new task transition blocks is partially disabled.

To enable the strong task transition check
➤ Change to the MATLAB Command Window and enter

rti_option TaskTransitionCheck Strong

In this mode, only the task transition blocks from RTI’s TaskLib are allowed for the data transfer from or to interrupt-block-driven tasks. Whenever Simulink finds a direct data connection between different tasks, it issues an error message that points to the incorrect connection. For detailed information on migrating a model created with RTI 4.2 or earlier to the new task transition blocks, see Migrating to RTI 4.4 in dSPACE Release New Features and Migration of dSPACE Release 3.5.

How to Use Hardware Interrupts

RTI provides hardware interrupt blocks for the different dSPACE boards. There are various types of hardware interrupts depending on the type of board you consider, for example, external interrupts used for the emergency stop button in the above-mentioned example. For details on the different hardware interrupts, refer to the corresponding reference material.
To use hardware interrupts

1. Place the **hardware interrupt** block (see example on the left) from the appropriate I/O library in your Simulink model. This block is the trigger source for the function-call subsystem.

2. Do not place this block inside a triggered subsystem or an enabled subsystem. It always activates its function-call subsystem regardless of whether its parent subsystem is active or not.

2. Open the Block Parameters dialog, and specify the settings you require, for example, the interrupt type.

   If you use a modular system, you also have to specify the correct board number.

3. Connect the outport of the trigger source to the trigger port of the function-call subsystem.

   See *How to Prepare a Function-Call Subsystem* on page 81 for instructions on how to prepare a function-call subsystem.

   Do not connect multiple function-call subsystems to the same trigger source.

   You may include the following blocks in the connection:

   **Mux/Demux, Inport/Outport, and From/Goto**.

   **RTI-MP** To transfer an interrupt from one CPU to another, you have to connect the trigger source and the function-call subsystem via an IPI block. No other blocks are allowed for this connection. The IPI block must list all target CPUs involved in the execution of the function-call subsystem. However, for local interrupts you do not need to include the IPI block in the connection. See *How to Implement Interprocessor Interrupts* on page 147 for details on IPI blocks.

   For multiprocessor systems, you can also assign different parts within the subsystem to different CPUs: Just add the required IPC blocks and/or Default CPU blocks (see *How to Implement Interprocessor Communication* on page 129 and *How to Define a Default CPU* on page 145).
For a demo of this task type, see the Hardware Interrupt demo model, which you can access via the RTI block library – Demos – TaskLib Demos.

Hardware interrupts from a timer device

Each dSPACE controller/processor board provides a number of timer devices. By default, RTI implicitly uses one of these to trigger the timer task(s) of the model.

Via the Timer Interrupt block from RTI’s TaskLib, you can use the free timer devices of the controller/processor board to periodically trigger subsystems in your model. The resulting tasks are not synchronized to the fixed step size of the model and are not tied to the rate-monotonic priority scheme of the model. For example, you can create a periodic task with a very slow sampling rate but a priority higher than the fastest timer task, or create a task with a sampling rate that is not a multiple of the fixed step size.

Tasks driven by the Timer Interrupt block are executed periodically but behave like tasks driven by interrupt blocks.

For a demo of this task type, see the Timer Interrupt demo model, which you can access via the RTI block library – Demos – TaskLib Demos.

How to Use Software Interrupts

In some cases you might want to trigger the execution of a task by means of a Simulink signal that does not have the function-call type. However, it is not possible to connect a non function-call Simulink signal to the trigger port of a function-call subsystem. RTI provides the Software Interrupt block, which generates an interrupt signal depending on the enable/trigger signals and lets you achieve very flexible interrupt handling.
To use software interrupts

1. Place the **Software Interrupt** block from RTI's TaskLib in your Simulink model.
   You may place this block inside any subsystem, for single-processor systems also inside triggered and enabled subsystems.

2. Open the Block Parameters dialog of the Software Interrupt block, and select the desired trigger/enable port.

3. Connect the Simulink signal(s) you want to use as the trigger/enable to the trigger/enable port(s) of the **Software Interrupt** block.

4. Connect the outport of the trigger source to the trigger port of the function-call subsystem.
   See *How to Prepare a Function-Call Subsystem* on page 81 for instructions on how to prepare a function-call subsystem.
   Do not connect multiple function-call subsystems to the same trigger source.
   You may include the following blocks in the connection: **Mux/Demux, Inport/Outport**, and **From/Goto**.

**RTI-MP** To transfer an interrupt from one CPU to another, you have to connect the trigger source and the function-call subsystem via an IPI block. No other blocks are allowed for this connection. The IPI block must list all target CPUs involved in the execution of the function-call subsystem. However, for local interrupts you do not need to include the IPI block in the connection. See *How to Implement Interprocessor Interrupts* on page 147 for details on IPI blocks.

For multiprocessor systems, you can also assign different parts within the subsystem to different CPUs: Just add the required IPC blocks and/or Default CPU blocks (see *How to Implement Interprocessor Communication* on page 129 and *How to Define a Default CPU* on page 145).
For a demo of this task type, see the Software Interrupt demo model, which you can access via the RTI block library – Demos – TaskLib Demos.

**Example of a Task Driven by a Hardware Interrupt Block**

Consider the throttle control example (see Throttle Control Example on page 20). To close the throttle valve when the emergency stop button is operated, the emergency stop button has to generate a hardware-interrupt signal. The following illustration shows the necessary changes:

However, the interrupt signal does not need to come from the hardware. If you want to generate an interrupt signal by the Simulink model itself, you can use a software interrupt.
In the example shown above, the hardware interrupt sets just the output of the function-call subsystem to zero. However, the Switch and DIO_TYPE1_PWM_FP_M1_C1 blocks are processed in the timer task, which delays the output signal until the next execution step of the timer task.
Assigning Parts of the Model to the Background Task

The Background block allows you to utilize the background task for subsystems that are uncritical with regard to timing. This task is not triggered by interrupts but executes whenever no other task needs to be executed.

**Basics**

If you want to utilize the background task in your model, you should know its characteristics. Refer to *Basics of the Background Task* on page 93.

**Assigning**

The part of the model you want to assign to the background task must be located in a function-call subsystem that is connected to the Background block. Refer to *How to Assign Parts of the Model to the Background Task* on page 94.

For an example on using the background task for model calculations, refer to *Example of a Subsystem Assigned to the Background Task* on page 95.

**Basics of the Background Task**

The background task is not accessible in the RTI Task Configuration dialog because it does not have a priority. Therefore, you cannot predict when or how often it is executed. The calculations made in it might be either distributed over several base sampling steps or executed several times between two base sampling steps. There is no Task Info group in the variable description file, therefore `turnaroundTime`, `taskCallCount`, etc. are not available for the background task.

The background task remains active no matter what the current simulation state is (unless you stop the real-time processor or reset the hardware). If you want the connected function-call subsystem to be conditionally executed whenever a specific simulation state is active, you can specify the desired condition (`RUN`, `PAUSE`, `STOP`) in the Block Parameters dialog of the Background block.
If desired, you can use several instances of the Background block in a single model or subsystem.

**How to Assign Parts of the Model to the Background Task**

**To assign a part of the model to the background task**

1. Place the **Background** block from RTI's TaskLib in your model. This block is the caller for the function-call subsystem.

2. Open the Block Parameters dialog of the Background block and specify the desired condition(s) (**RUN**, **PAUSE**, **STOP**) on which the subsystem should be executed in the background task.

3. Connect the output port of the Background block to the trigger port of the function-call subsystem (see **How to Prepare a Function-Call Subsystem** on page 81 for instructions on how to prepare a function-call subsystem).

   - Do not connect multiple subsystems to the same Background block.
   - With single-processor systems, you may include the following blocks in the connection: **Mux/Demux**, **Inport/Outport**, and **From/Goto**.
   - With multiprocessor systems, no blocks are allowed in this connection.

For multiprocessor systems, you can also assign different parts within the subsystem to different CPUs. Just add the required IPC blocks. Make sure not to use the Synchronized Swinging Buffer protocol.

To assign the entire subsystem to a single CPU, add a Default CPU block or place the entire subsystem together with its trigger source inside another parent subsystem that is assigned to a single CPU (see **How to Define a Default CPU** on page 145).
If you use the Background block extensively, the update of ControlDesk's virtual instruments might become slower because the corresponding host service for ControlDesk is executed in the background task as well.

Example of a Subsystem Assigned to the Background Task

Consider the throttle control example (see Throttle Control Example on page 20) and suppose you want to monitor the flow rate. This model part has nothing to do with the actual control problem, so its timing is not critical. Especially if the CPU load is rather high, you might want to place such uncritical subsystems in the background task so that they are executed only if the CPU is idle. A possible model design for the example is shown below:
How to Change the Task Priorities

If you want to change the general execution order of your real-time application, you can rearrange the priorities of its tasks.

**Basics**

Beforehand, you have to specify whether you want to execute your model in the single timer task mode or multiple timer task mode. Depending on the execution mode chosen, a different constellation of timer tasks results. The priority of a timer task is fixed in relation to the priorities of the other timer tasks. This is due to the rate-monotonic priority scheme. For detailed information on the execution modes refer to *Timer Tasks* on page 63.

Consider the throttle control example (see *Throttle Control Example* on page 20). Suppose you have included both the emergency stop button and the monitoring of the flow rate. If you selected the multiple timer task mode for this model, the RTI Task Configuration dialog shown below lists the following distribution of priorities:
RTI Task Configuration dialog

The RTI Task Configuration dialog shows all the tasks of your model in decreasing order of their priorities (highest priority on top). The entries of the task list show the priority number, followed by the name of the task. The priority number can range from 1 as the highest priority to 127 as the lowest priority. The background task has no priority; thus, can be interrupted by any other task and it is not displayed in the RTI Task Configuration dialog. RTI derives the name of a task as follows:

- For timer tasks from the sample time and offset (if any), for example, Timer Task 2 [1] s
- For tasks driven by interrupt blocks from the name of the driving Timer, Hardware or Software Interrupt block

The Task configuration frame displays the task configuration details for the task that is currently highlighted in the task list.

In some cases, inappropriate distribution of priorities might lead to overrun situations – and even to deadlock for multiprocessor systems.

Rearranging priorities

To rearrange the priorities of a model

1. Open the RTI Task Configuration dialog:
   - For single-processor systems you can access it from the Real-Time Workshop page of the Simulation Parameters dialog (category “RTI simulation options”).
   - For multiprocessor systems you can access it from the individual CPUs’ pages of the Multiprocessor Setup dialog.

2. You can alter the priority of one task in relation to another and change its position in the task list via the Up, Down and As Previous buttons.

If you modified the number or settings of interrupt blocks in your model, the RTI Task Configuration dialog opens automatically during automatic program building so that you can confirm the task configuration.

For details on the RTI Task Configuration dialog, refer to RTI Task Configuration Dialog in the RTI and RTI-MP Implementation Reference.
Handling Tasks

Non real-time simulation mode
For testing purposes you might want to run the real-time program in a non real-time mode. If you use the as-fast-as-possible simulation mode, you should assign the lowest priority to the timer task. Otherwise virtually no time would be left for the lower-priority tasks driven by interrupt blocks because the timer task is called at a very fast pace in the background task of the model. For details on the non real-time execution modes, refer to Real-Time Workshop Page (for RTI) and Main Page (for RTI-MP) in the RTI and RTI-MP Implementation Reference.

Displaying the task configuration
Once your model is running on the dSPACE hardware, you can find the task configuration (priority, overrunCheckType, etc.) of each task in the Task Info group, which you can access via ControlDesk. Refer to Creating Data Connections in the ControlDesk Experiment Guide for details on connecting a variable to a ControlDesk instrument.
How to Handle Overrun Situations

You can choose the desired overrun handling individually for each task via the RTI Task Configuration dialog. This is also displayed by the overrunCheckType variables of the real-time application (available via the Task Info groups in ControlDesk’s Variable Browser).

**Available overrun strategies**

- **Stop simulation**  
  This setting stops the application whenever an overrun occurs.

- **Queue task before simulation stop**  
  This setting counts overruns and schedules the new task call. You can specify a maximum number of queued task calls. If this limit is reached, the application is stopped.

- **Ignore overrun**  
  This setting counts the overruns but does not schedule the new task call in case an old instance is still running.

Even though stopping the application due to an overrun means that a run-time error has occurred, the application remains in the simulation state STOP and can be restarted (see **How to Set and Read the Simulation State** on page 197 – for details on run-time errors, see **Reaction to Run-Time Errors** on page 208).

**Configuring overrun strategies**

**To configure the overrun strategy for a task**

1. Open the RTI Task Configuration dialog:
   - For single-processor systems you can access it from the Real-Time Workshop page of the Simulation Parameters dialog (category “RTI simulation options”).
• For multiprocessor systems you can access it from the individual CPUs’ pages of the Multiprocessor Setup dialog.

The RTI Task Configuration dialog opens:

2 In the task list, highlight the task you want to modify.

3 Select the desired overrun strategy in the Overrun strategy frame.

   If you select the queue task before simulation stop setting, you also have to specify the number of max. queued task calls.

Analyzing overrun situations

You can analyze any overrun situations you encounter by means of the following variables:

The overrunCount variable of a task counts the total number of overruns that occurred for it.

The overrunQueueCount variable of a task is incremented for each queued task call and decremented each time a task call finishes execution.

The taskCallCount variable of a task is incremented whenever the task is called and executed. It does not include the task calls that caused overrun situations.

These variables can be displayed in ControlDesk (refer to the variable descriptions in the RTI and RTI-MP Implementation Reference).
How to Combine Several Interrupt Sources

Consider the throttle control example (see Throttle Control Example on page 20). Suppose you want to close the throttle valve when someone presses the emergency stop button or if the flow rate in the throttle valve exceeds a certain limit. You have to combine these interrupts via a Mux block to trigger a single function-call subsystem as shown in the illustration below:

Via the Mux block, each interrupt source can then create its own task from the function-call subsystem, which therefore might execute in multiple calls. Since each call has its own overrun check, no overrun situation occurs – provided each call is triggered by a different interrupt. If an interrupt with high priority occurs, it suspends the call executed as a task with a lower priority and creates a new call.
In combining interrupts to trigger the same subsystem, the following limitations apply:

- The code generated for the function-call subsystem is not safe on reentrance. That is, the data of the first call of a task is lost if another call starts. This is due to the global definition of block outputs in the function-call subsystem. Therefore, if two tasks act simultaneously on the same function-call subsystem, you cannot predict which block outputs will result. Depending on the blocks calculated in the task, even serious memory corruption can result. For this reason you should use identical priorities when combining interrupt sources: Tasks of the same priority do not interrupt each other.

- You cannot combine the function-call signals from RTI interrupt blocks with those of other blocks, for example, a Function-Call Generator block or Stateflow chart.

- You cannot use the ssEnableSystemWithTid and ssDisableSystemWithTid macros to enable and disable a function-call subsystem driven by an RTI interrupt block because the subsystem will always be executed when it is triggered by an RTI interrupt block.

- With RTI-MP you can combine only local interrupts to trigger the same subsystem, i.e., all interrupts must originate from the same CPU. None of the function-call signal lines may contain an IPI block.

To combine several interrupt sources

1. Place a Mux block from the Simulink library on the same level as the function-call subsystem you want to trigger.
2. Connect all Timer, Hardware or Software Interrupt blocks that you want to use as the triggers to the input ports of the Mux block.
3. Connect the output port of the Mux block to the trigger port of the function-call subsystem.
4. Assign the same priority to all the resulting tasks.
If you wish to detect which interrupt source triggered the function-call subsystem, carry out the additional instructions provided below.

**To detect the trigger source inside the function-call subsystem**

1. Open the function-call subsystem.
2. Select the **Show output port** checkbox in the Block Parameters dialog of the subsystem's Simulink **Trigger** block.

The Simulink Trigger block then provides a vectorized (multiplexed) signal in which all vector elements show zero, except for the one that triggered the function-call subsystem.
How to Subschedule Tasks

Consider the throttle control example (see Throttle Control Example on page 20). Suppose you want two actions to be carried out in a specific sequence (called subscheduling) after the emergency-stop button was pressed. You can achieve this behavior by means of additional priorities, which you can define via Software Interrupt blocks as illustrated below. The blocks forming the actual control loop have been left out for clarity.

For this approach to function correctly, the interrupt block of the master subsystem must have a higher priority than the Software Interrupt blocks used for the subscheduling. Otherwise the behavior is unpredictable because the execution order of the Software Interrupt blocks (in the master subsystem) is not predefined.
The following RTI Task Configuration dialog shows a valid distribution of the priorities:

With RTI-MP, subscheduling is possible only for local interrupts, i.e., if the source and target CPUs are identical.

To subschedule a task

1. Build up the level triggered by the **hardware interrupt**. This acts as the master subsystem.
2. Place as many **Software Interrupt** blocks from RTI’s TaskLib in the master subsystem as you require for subscheduling.
3. Add the same number of function-call subsystems from RTI’s TaskLib to the model. These act as slave subsystems.
4. Connect the slave subsystems to the **Software Interrupt** blocks of the master subsystem.

You may include the following blocks in the connection: **Mux/Demux, Inport/Outport, and From/Goto**.
Basics of Time-Triggered Tasks and Timetables

In contrast to event-based execution of tasks, this blockset lets you specify time-triggered task execution. Time-triggered tasks and timetables let you implement task trigger times and even complete time-dependent schedules (i.e., timetables) for tasks. You should read up on the basics of the two techniques.

Basics

Time-triggered tasks and timetables are features of RTI and RTI-MP that let you schedule one or more tasks with predefined delay times in relation to a single trigger event.

Via time-triggered tasks and timetables, you can implement more flexible task handling in your model, for example, for time-based data processing.

Suppose you want to take a measurement n milliseconds after the ignition of an engine. The value of n depends on several parameters such as engine speed and acceleration. This can be implemented via a time-triggered task. To start several measurements, you can implement a timetable.

You can include multiple independent time-triggered tasks and/or timetables in your model.

Restrictions

- The total number of time-triggered tasks and timetable tasks that can be simultaneously scheduled is limited to 31 (for DS1103) or 127 (for DS1005, DS1006, and DS1401).
- Not available for DS1104 platform support.

Time-triggered tasks

When a time-triggered task is triggered (for example, by a hardware interrupt), the actual task is started with a trigger delay time.

The trigger delay time is specified via a Simulink signal, which may vary during the simulation. If you specify a vector as the trigger delay time, the actual task is started once for each vector element.
The following illustration shows the timing for a time-triggered task with three trigger delay times having the values 0.1 ms, 0.2 ms and 0.5 ms on occurrence of the trigger event:

![Diagram showing task trigger delay times: 0.5 ms, 0.2 ms, 0.1 ms](image)

**Timetables**

A timetable consists of a set of timetable tasks, each of which has its own predefined trigger delay time (scalar or vector). In addition, the entire timetable has a common timetable start delay time.

When the timetable is triggered (for example, by a hardware interrupt), each timetable task is started with its effective trigger delay time, i.e., timetable start delay time (variable) + predefined trigger delay time of the timetable task (predefined and fixed).

The timetable start delay time is specified via a scalar Simulink signal, which may vary during the simulation.
The following illustration shows the timing for a timetable with two timetable tasks: Task A has a trigger delay time of 2 ms and Task B has the trigger delay times 1 ms and 4 ms. The timetable start delay time is 1.5 ms on occurrence of the trigger event. Therefore, the effective trigger delay times are 3.5 ms for Task A, and 2.5 ms and 5.5 ms for Task B.

**Self-triggered task**

If desired, you can implement a self-triggered task or timetable by using a time-triggered task or timetable task as trigger event for the task or timetable itself. To initiate the first trigger event, you must use the simulation start as a trigger event by selecting the **Execute time-triggered task once at simulation start** option in the **Time-Trigger Set** block or the **Start timetable once at simulation start** option in the **Timetable Start** block.

- When a time-triggered task is started at simulation start, its task trigger delay time is ignored for the first execution step and the actual task is started immediately.

- When a timetable is started at simulation start, its timetable start delay time is ignored for the first execution step and the timetable tasks are started with only their individual task trigger delay times.
The following illustration shows the timing for a time-triggered task and a timetable that use the simulation start (T=0) as trigger event. The task trigger delay time(s) of the time-triggered task and the timetable start delay time for the timetable tasks are ignored for the first execution step.

**Executing the timer tasks as time-triggered or timetable tasks**

You can execute the timer tasks of your model as time-triggered tasks or timetable tasks by connecting the Time-Triggered Task block or Timetable Task block to the Timer Task Assignment block.

Suppose you want to implement an application that performs data sampling at a fast sampling rate and feeds the sampled data vectors to the timer tasks for further data processing. In this case you can implement two time-triggered tasks with appropriate trigger delays. Connect the Timer Task Assignment block to one of the time-triggered task blocks. This time-triggered task must be configured to use the simulation start as trigger event.
How to Implement Time-Triggered Tasks

Time-triggered tasks let you start a task with one or more trigger delay times. The trigger delay time is a Simulink signal.

**Basics**
You should read up on the basics of time-triggered tasks. Refer to Basics of Time-Triggered Tasks and Timetables on page 106.

A time-triggered task is implemented by one Time-Trigger Set block and one Time-Triggered Task block, which are associated by means of a unique identifier (time-trigger ID).

- When the Time-Trigger Set block is executed, it sets the trigger delay time for its task according to the signal at the Trigger delay inport. When the trigger delay time is reached the task is scheduled.

- The actual time-triggered task consists of a function-call subsystem that is connected to the Time-Triggered Task block.

**Prerequisites**
Prepare a function-call subsystem as described in How to Prepare a Function-Call Subsystem on page 81.

**Restrictions**
- The total number of time-triggered tasks and timetable tasks that can be simultaneously scheduled is limited to 31 (for DS1103) or 127 (for DS1005, DS1006, and DS1401).

- Not available for DS1104 platform support.

**Instructions**
To implement a time-triggered task

1. Place the Time-Trigger Set block from RTI’s TaskLib/Time-Trigger library in the model. You can place it anywhere but in the background task.

   Open the Block Parameters dialog of the Time-Trigger Set block and enter a unique Time-Trigger ID.

   Close the dialog.
To implement a self-triggered task, place the block in the function-call subsystem driven by the associated Time-Triggered Task block. In this case, select the **Execute time-triggered task once at simulation start** checkbox.

2. Connect the desired Simulink signal to the **Time-Trigger Set** block, which serves as the task trigger delay time. To specify more than one task trigger delay time, use a vector signal.

3. Place the **Time-Triggered Task** block from RTI's TaskLib/Time-Trigger library in your model. This block is the trigger source for the time-triggered task.

   Do not place this block inside a triggered subsystem or an enabled subsystem. It always activates its function-call subsystem regardless of whether its parent subsystem is active or not.

   Open the **Block Parameters** dialog of the Time-Triggered Task block. Enter the same **Time-Trigger ID** as in step 1.

   Close the dialog.

4. Connect the outport of the trigger source to the trigger port of the function-call subsystem.

   See **How to Prepare a Function-Call Subsystem** on page 81 for instructions on how to prepare a function-call subsystem.

   Do not connect multiple function-call subsystems to the same trigger source.

   You may include the following blocks in the connection: **Mux/Demux, Inport/Outport, and From/Goto**.

   **RTI-MP** To transfer an interrupt from one CPU to another, you have to connect the trigger source and the function-call subsystem via an IPI block. No other blocks are allowed for this connection. The IPI block must list all target CPUs involved in the execution of the function-call subsystem. However, for local interrupts you do not need to include the IPI block in the connection. See **How to Implement Interprocessor Interrupts** on page 147 for details on IPI blocks.
For multiprocessor systems, you can also assign different parts within the subsystem to different CPUs: Just add the required IPC blocks and/or Default CPU blocks (see How to Implement Interprocessor Communication on page 129 and How to Define a Default CPU on page 145).

To execute the timer tasks of the model as time-triggered tasks, connect the Timer Task Assignment block to the outport of the Time Triggered Task block.
How to Implement Timetables

A timetable lets you start a set of timetable tasks in relation to a single trigger event. The start delay time for the timetable is a Simulink signal.

**Basics**

You should read up on the basics of timetables. Refer to Basics of Time-Triggered Tasks and Timetables on page 106.

A timetable is implemented by one Timetable Start block and at least one Timetable Task block, which are associated by means of a unique identifier (timetable ID).

- When the Timetable Start block is executed, it triggers the timetable with the timetable start delay time according to the signal at the Start delay inport.

- Each timetable task consists of a function-call subsystem that is connected to a Timetable Task block. When the timetable start delay time plus the task trigger delay time of a Timetable Task block is reached, the actual task of that Timetable Task block is executed.

**Prerequisites**

Prepare as many function-call subsystems as needed for timetable tasks. Refer to How to Prepare a Function-Call Subsystem on page 81.

**Restrictions**

- The total number of time-triggered tasks and timetable tasks that can be simultaneously scheduled is limited to 31 (for DS1103) or 127 (for DS1005, DS1006, and DS1401).

- Not available for DS1104 platform support.

**Instructions**

To implement a timetable task

1. Place the Timetable Start block from RTI’s TaskLib/Time-Trigger library in the model. You can place it anywhere but in the background task.

   Open the Block Parameters dialog of the Timetable Start block and enter a unique Timetable ID.

   If you want to specify a start delay time for the timetable, select the Enable timetable start delay inport checkbox.

   Close the dialog.
To implement a self-triggered task, place the block in the function-call subsystem driven by an associated Timetable Task block. In this case, select the **Start timetable once at simulation start** checkbox.

2 If you selected the **Enable timetable start delay import** checkbox, connect the desired Simulink signal to the **Timetable Start** block. The signal must be a scalar.

3 Place the **Timetable Task** block from RTI’s **TaskLib/Time-Trigger** library in your model. This block is the trigger source for the timetable task.

   Do not place this block inside a triggered subsystem or an enabled subsystem. It always activates its function-call subsystem regardless of whether its parent subsystem is active or not.

   Open the **Block Parameters** dialog of the Timetable Task block.

   Enter the same **Timetable ID** as in step 1.

   Enter the desired **Task trigger delay time**. To specify more than one task trigger delay time, specify a vector.

   Close the dialog.

4 Connect the outport of the trigger source to the trigger port of the function-call subsystem.

   See **How to Prepare a Function-Call Subsystem** on page 81 for instructions on how to prepare a function-call subsystem.

   Do not connect multiple function-call subsystems to the same trigger source.

   You may include the following blocks in the connection: **Mux/Demux**, **Inport/Outport**, and **From/Goto**.

**RTI-MP** To transfer an interrupt from one CPU to another, you have to connect the trigger source and the function-call subsystem via an IPI block. No other blocks are allowed for this connection.

   The IPI block must list all target CPUs involved in the execution of the function-call subsystem. However, for local interrupts you do not need to include the IPI block in the connection. See **How to Implement Interprocessor Interrupts** on page 147 for details on IPI blocks.
For multiprocessor systems, you can also assign different parts within the subsystem to different CPUs: Just add the required IPC blocks and/or Default CPU blocks (see How to Implement Interprocessor Communication on page 129 and How to Define a Default CPU on page 145).

To execute the timer tasks of the model as a timetable task, connect the Timer Task Assignment block to the outport of one of the Timetable Task blocks.

5 Repeat steps 3 … 4 as often as required to create the desired timetable.
Distributing the Model for MP Systems

As shown in the Introduction to RTI and RTI-MP on page 17, real-time simulation can be very demanding in terms of computing power – this is especially true for rather large models. If the computing power of single-processor systems does not meet your requirements you can use a dSPACE multiprocessor system. This allows you to break your large model into several smaller parts and to assign these parts to different CPUs. Thus, you can gain enough computing power to calculate even large models in real-time.

Consider the throttle control example (see Throttle Control Example on page 20) and suppose it would require more computing power than a single processor system can provide. You could then divide the model and assign the parts to the different CPUs of a multiprocessor system so that one board simulates the plant and the other board the controller. To get the desired multiprocessor model you have to add a Multiprocessor Setup block that holds the configuration parameters for multiprocessor models. You also have to modify all connection lines.
Distributing the Model for MP Systems

between the plant and controller so that they are passed through Interprocessor Communication blocks (IPC blocks). Hence, the IPC blocks mark the boundaries between the different CPUs. The following illustration shows the modified model for a multiprocessor system:

Configuring the multiprocessor model
The model needs to be configured for use on a multiprocessor system. Refer to How to Configure an RTI-MP Model on page 120.

Network topology
Consider a standard network topology for your multiprocessor models so that you can run them on the same hardware without reconfiguration. Refer to Standard Network Topology on page 122.

Interprocessor communication
Interprocessor communication allows data transfer from one CPU to another. Refer to Interprocessor Communication on page 123.

Interprocessor interrupts
Interprocessor interrupts allow tasks driven by interrupt blocks to be synchronized across various CPUs. Refer to Interprocessor Interrupts on page 147.

Beneficial model distribution
The efficiency of a multiprocessor application depends on the model distribution. Refer to Model Distribution Guidelines on page 151.
Related Topics

Optional CPUs

The optional CPU feature of RTI-MP allows you to:

■ Specify certain CPUs – and the corresponding real-time applications – as optional in RTI-MP
■ Work with topological subsets of the currently registered multiprocessor system in ControlDesk

This allows you to download a multiprocessor application either to the entire multiprocessor system or to a topological subset of it without having to rebuild the application.

Refer to Working with Subsets of a Multiprocessor Topology in the Features document of your board for details on this technique. You can find a demo model (Optional CPU for dual DS100<x> System demo) in the RTI-MP Demo Library for DS100<x> MP Systems.

Migration from AlphaCombo

The AlphaCombo system is no longer supported. You can migrate your AlphaCombo models either to a multiprocessor system or to a single-processor system based on the DS1005 or DS1006. Refer to How to Migrate from AlphaCombo to DS1005 or DS1006 on page 234.
How to Configure an RTI-MP Model

The Multiprocessor Setup dialog (available via the Multiprocessor Setup block) holds the majority of the global simulation parameters for the multiprocessor model and all the local options for each CPU, including the name for each CPU and its submodel. However, if you simulate the RTI-MP model in Simulink you must specify the desired simulation parameters via the Simulation Parameters dialog instead.

If you implement interprocessor communication to a new CPU, the corresponding CPU's page is automatically added to the Multiprocessor Setup dialog, where you can edit the properties of the new CPU and the name of the new submodel according to your requirements.

RTI-MP automatically selects the CPU with the name master as the master CPU. If no CPU is named master, RTI-MP selects the CPU with the name that is the first in the alphabet. The master CPU is marked with an (M) on the Main page of the Multiprocessor Setup dialog.

The board names used by ControlDesk's Platform Manager and by RTI-MP must be identical.

To configure a model for RTI-MP

1. Drag and drop the Multiprocessor Setup block from the RTI-MP Blocks library onto the root level of your model.

   If you double-click the Multiprocessor Setup block in the RTI-MP Blocks library, a preconfigured multiprocessor model is created. It contains a Multiprocessor Setup block and two IPC blocks.

2. Whenever you create a communication connection to a new CPU, you have to adapt the name of the new submodel on the corresponding CPU's page of the Multiprocessor Setup dialog (this CPU's page has the same name as the new CPU that you specified for the communication connection).
3 To build and download the RTI-MP model, you have to adapt all the settings of the Multiprocessor Setup dialog according to the instructions given in How to Specify Options for the Build Process on page 245.

4 Ensure that the GigaLink topology is defined via the Multiprocessor Topology Setup dialog of the model (see Multiprocessor Topology Setup Dialog in the RTI and RTI-MP Implementation Reference).

To make a slave CPU optional, select the Treat as optional CPU checkbox on the CPU’s page of the Multiprocessor Setup dialog (see CPU’s Page in the RTI and RTI-MP Implementation Reference and Working with Subsets of a Multiprocessor Topology in the Features document of your board).
Standard Network Topology

You might want to use a standard network topology for your multiprocessor models so that you can run all models on the same hardware without reconfiguration. The following illustration shows you a solution for connecting up to five boards so that each board has a direct connection to all other boards:
Interprocessor Communication

To simulate your model on a dSPACE multiprocessor system, you have to define the necessary interprocessor communication between the model parts you want to assign to the different CPUs. By implementing interprocessor communication you implicitly specify which model part belongs to each CPU.

For dSPACE systems, the different tasks of a real-time application require separate communication channels due to different timing characteristics. However, it is possible to group all data of a specific sample time and send it over just one communication channel. The same is true for the data of a task driven by an interrupt block that is executed on multiple CPUs. Usually, you need to transfer data of a single sample time only, so you do not need to use several communication channels.

**Connection type**
The different processor boards of a multiprocessor system communicate by means of GigaLink connections. Refer to [GigaLink Connection](#).

**Protocols**
RTI-MP supports two communication protocols for different purposes. You can select either protocol individually for each communication channel.

- [Swinging Buffer Protocol](#)
- [Virtual Shared Memory Protocol](#)

**Implementing**
To implement interprocessor communication, you need to specify the communication connection and channel, and configure them. Refer to [How to Implement Interprocessor Communication](#).

**Direct feedthrough**
Direct feedthrough can decrease the overall performance and even cause deadlocks. Refer to [Model Evaluation](#).

**Buffering and downsampling**
A communication channel can be operated with the sample time of the involved CPUs or less often. The CPUs can also have different sample times. Refer to [Communication with a Sample Time Transition](#).
Distributing the Model for MP Systems

**Default CPU**

Define a default CPU for model parts that do not use interprocessor communication. Refer to *How to Define a Default CPU* on page 145.

**GigaLink Connection**

Several DS1005 or DS1006 boards can communicate by means of the DS910 or DS911 GigaLink Modules respectively. The DS91<sub>x</sub> GigaLink Module provides four bidirectional GigaLinks that allow a flexible network topology.

RTI-MP supports multiprocessor systems that consist of either only DS1005 or only DS1006 boards.

Each GigaLink provides 16 data channels per direction and each channel can transfer several signals (up to 8 KB). For details, see *DS910 GigaLink Module* in the *DS1005 Features* document or *DS911 GigaLink Module* in the *DS1006 Features* document. RTI-MP uses logical communication channel numbers (for example, master to slave on channel 1) and manages the mapping to the available physical communication channels. Throughout this Implementation Guide the term *communication channel* refers to the logical channels, as can be seen in the IPC blocks of the Simulink model.

The `<model>_tp.m` file holds the information about the existing processor network topology and needs to be updated whenever you change the physical GigaLink connections of your multiprocessor system. This can be done via the Multiprocessor Topology Setup dialog (see *Multiprocessor Topology Setup Dialog* in the *RTI and RTI-MP Implementation Reference*). Via this dialog you can also set up a virtual network topology, for example, if the hardware is not available yet.
The following illustration shows a communication channel between two processor boards, which are connected via the GigaLinks 1 (sender) and 2 (receiver). RTI-MP automatically maps the logical channel numbers in the IPC block (for example, master to slave on channel 1) to this physical connection so that the Simulink model does not contain any hardware-dependent information. Therefore, in the Simulink model you cannot see which physical connections are used for interprocessor communication. In this example, physical channel 0 is used.

Swinging Buffer Protocol

This protocol defines three buffers between the sender and receiver, which can hold a single data vector each. The sender and receiver allocate and lock a separate buffer for writing or reading data. Because of this locking, there is no risk of data inconsistency for this protocol.

The following illustrations show how the swinging buffer protocol works.
- $t[i]$: At the beginning the sender fills the first buffer.

![Diagram showing buffer protocol at time $t[i]$]

When the sender finishes writing it unlocks its buffer and locks the next one. The receiver reads the previously written data.

- $t[i+1]$: The procedure described above is repeated using the next buffer: The sender fills its current buffer after which the receiver reads it and so on.

![Diagram showing buffer protocol at time $t[i+1]$]

In order to reduce unnecessary waiting times, RTI-MP computes the output signals of a submodel as early as possible. It also carries out all computations that are independent of input signals before reading the input signals, which in turn are read as late as possible.

**Synchronized and unsynchronized**

The swinging buffer protocol is available in synchronized and unsynchronized mode:
Synchronized  If no new data is available the receiver waits for the sender to deliver new data. Data consistency and synchronized data transfer are especially important in control applications where a delay might cause stability problems.

Unsynchronized  If no new data is available the receiver reads the data vector of the previous sampling step. As a result there are no waiting times but the signal is delayed by one sampling step. Depending on the characteristics of the application, usually one of the following two cases applies:

- The receiver always gets the current buffer. This behavior is identical to the synchronized swinging buffer mode and is described above.
- The receiver always gets the buffer that comes from the previous sampling step. This behavior corresponds to the synchronized swinging buffer mode with an additional Unit Delay block. This is shown in the next illustration.

In some cases the synchronized swinging buffer protocol might cause overrun or deadlock situations when the receiver waits for new data. For details on how a deadlock situation can arise, see Model Evaluation on page 133.

If you want to avoid waiting times due to interprocessor communication you can select the unsynchronized swinging buffer protocol or virtual shared memory protocol. However, simulation robustness might decrease with these protocols, because they do not provide synchronized data transfer.
Virtual Shared Memory Protocol

This protocol defines only one buffer for the data vector to be transmitted. Neither the sender nor the receiver locks the buffer but accesses it as if it were its local memory. There is no synchronization between the sender and receiver, so the receiver might read the same data vector several times or it might not catch each new data vector. Because writing and reading can be simultaneous, this protocol might result in data inconsistency, meaning that the input buffer (as seen by the receiver) contains data resulting from different sampling steps of the sender. The following illustration shows how the virtual shared memory protocol works:
How to Implement Interprocessor Communication

The IPC block defines the boundary between the different CPUs. Its Block Parameters dialog, the Communication Channel Setup dialog, is shown below:

Adding IPC blocks

For each communication channel, you can specify the protocol to be used, the step sizes and whether you want to operate the channel in buffered/unbuffered mode (see Swinging Buffer Protocol, Virtual Shared Memory Protocol and Communication with a Sample Time Transition on page 137).

You can also specify the block width and the data type to be used. Technically, there is no difference if you connect several signal lines via one single IPC block (using multiple ports or a vectorized port) or via several separate blocks with the same channel number. The Communication Channel Setup dialog shows the total Number of signals for all Member IPC blocks.
To add interprocessor communication

1. Add as many IPC blocks as you need from the RTI-MP Blocks library to the root level of your model.

   You can also add IPC blocks to function-call subsystems that are driven by interprocessor interrupts or connected to the Background block. For details, refer to *How to Implement Interprocessor Interrupts* on page 147.

2. Open the Communication Channel Setup dialog for the different IPC blocks.

   The currently selected IPC block is highlighted in the model and its name in the Member IPC blocks frame of the Communication Channels Setup dialog is displayed in a bold font.

   You can easily locate the member IPC blocks via **Show Block**.

3. Specify the settings according to your requirements: for each member IPC block, specify the number of ports, the widths of the ports and the data type.

4. Check if the settings of the communication channel (the connection to be used, the protocol and the number of buffered samples) meet your requirements.

   If you specify a number greater than 1 for the buffered samples, you have to make all member IPC blocks use the double data type.
Change connection

You might want to make a single IPC block use a different communication channel. Carry out the following instructions:

To make a single IPC block use a different channel

1. Open the Communication Channel Setup dialog of the IPC block you want to modify.
2. Open the Change Connection dialog of the IPC block you want to modify.
3. Specify the desired Source CPU, Destination CPU and Channel for the IPC block.
   - The logical channel numbers are mapped to the channels of the available GigaLink connections via the network topology file.
   - If this communication channel already exists, the IPC block uses the existing channel and appears in the Member IPC blocks frame of the corresponding Communication Channel Setup dialog.
   - If this communication channel is new to the model, the IPC block uses a new channel without further Member IPC blocks.

If you add a new CPU to the model, you also need to configure the corresponding CPU’s page of the Multiprocessor Setup dialog (see How to Configure an RTI-MP Model on page 120).
Select CPUs for channel

You might want to redirect all member IPCs of a specific communication channel to a new source or destination CPU. In this case, carry out the following instructions:

To redirect all member IPCs of a channel to a new CPU

1. Open the Communication Channel Setup dialog of one of the member IPC blocks.

2. Enter the new settings for the desired Source CPU, Destination CPU and Channel in the Communication Channel frame. The logical channel numbers are mapped to the channels of the available GigaLink connections that are specified in the network topology file.

The communication channel is redefined according to your settings. If you enter a CPU that is new to the model, the corresponding CPU's page is added to the Multiprocessor Setup dialog.

If you add a new CPU to the model, you also need to configure the corresponding CPU's page of the Multiprocessor Setup dialog (see How to Configure an RTI-MP Model on page 120).

Suppose you want to use a third processor board (named slave_b) for a multiprocessor model (CPU names master and slave). Ensure that the Multiprocessor Setup dialog is closed. Copy an existing IPC block, and open its Change Connection dialog. Enter the name slave_b in the Source CPU or Destination CPU edit field. The Communication Channel Setup dialog is adapted accordingly, and in the current Simulink model, the IPC block displays the name of the new CPU. The next time you open the Multiprocessor Setup dialog, a new CPU's page named slave_b is available in this dialog.

The field for the application name is initially empty. Here you must insert the name of the object file to be generated. All other fields for the new CPU are initialized with the following default values: Step size multiple = 1 and Solver = ode1.
Connection, channel and IPC block

The following illustration shows the relationship between the Communication Connection, Communication Channel and IPC blocks. It summarizes the effects if you change the connection settings in the Communication Channel Setup dialog or in the Change Connection subdialog of an IPC block:

Model Evaluation

Whenever the submodel of a CPU is extracted from the overall multiprocessor model (for example, when building the model), RTI-MP replaces the IPC blocks of the overall model with either an IPC Input block or an IPC Output block – depending on whether the CPU receives or sends data via this IPC channel.

If a communication channel uses the Synchronized Swinging Buffer protocol, the receiving CPU must wait until new data is available before it can calculate the blocks depending on this data. Therefore RTI-MP computes the output signals of a submodel as early as possible so that unnecessary waiting times for other CPUs are avoided. For the same reason, RTI-MP reads the input signals as late as possible, meaning that all computations independent of the input data are done in advance until the CPU cannot continue the computation without input data from another CPU.
RTI-MP achieves this execution order by means of a high block priority for IPC Output blocks and a low block priority for IPC Input blocks. However, in the overall RTI-MP model, RTI-MP cannot assign the priorities until the submodels of the individual CPUs are extracted from the overall multiprocessor model. Whenever the submodel of a CPU is extracted from the overall multiprocessor model, RTI-MP uses the following two techniques to assign the highest priority to IPC Output blocks:

- The IPC Output blocks internally contain a level 2 S-function with the `SS_OPTION_PLACE_ASAP` option set. Simulink executes such S-functions as soon as possible in the overall execution order of the model – if possible, even before all other Simulink blocks.
- The IPC Output blocks are set to a priority value of -10,000, which represents a very high priority.
- The IPC Input blocks are set to a priority value of 10,000, which represents a very low priority.

The following example shows how a user-specific priority can cause a block priority violation in the extracted submodel while being unproblematic in the overall model.

Consider the demo model `pipt1_dual1005_sl.mdl` and suppose you assigned a priority value of 1 to Sum block `e`. This causes no priority conflict in the overall multiprocessor model – you can start the Simulink simulation without a problem:
However, in the extracted submodel `rtimp_sm_pi_1005.sl` this priority value conflicts with the RTI-MP-specific high priority of the IPC Output block (-10,000). Due to topological dependencies, Sum block `e`, which has a low priority, must be computed before the IPC Output. Therefore, Simulink issues a warning when you start the Simulink simulation or code generation for the submodel.

**Direct feedthrough**

Direct feedthrough terms can lead to waiting times and even cause deadlocks. Refer to [Direct Feedthrough on page 135](#).

**Checking block priorities**

The execution order of the individual blocks can be controlled via block priorities and level 2 S-functions. Refer to [Modifying the Block Execution Order on page 223](#).

A block priority violation can lead to warnings. Refer to [How to Check for Illogical Block Priorities on page 225](#).

**Direct Feedthrough**

If the signal path from one IPC to another does not contain any block that corresponds to a state-space description with zero D matrix (for example, integrator, unit delay, or memory) this signal path is called a direct feedthrough term. Its output is directly dependent on its input; thus, the output cannot be calculated before the input data is available. An example of such a direct feedthrough term is shown below:
In conjunction with the Synchronized Swinging Buffer protocol, such direct feedthrough terms can cause undesired waiting times, as shown below, where the master processor contains a direct feedthrough term:

To eliminate such waiting times you have to avoid the direct feedthrough terms, as shown below:

For this model there is no waiting time, as you can see in the following illustration. This is due to the outputs that do not depend directly on the inputs. The time that was freed up is colored in grey:
Direct feedthrough terms in conjunction with the Synchronized Swinging Buffer protocol can cause deadlock situations.

Consider the following model (with both IPC channels using the Synchronized Swinging Buffer protocol):

[Diagram of the model with signal flow]

In this model there are direct feedthrough terms for both CPUs. Such a signal path is called an algebraic loop and causes a deadlock situation because each CPU is waiting for input data from the other before being able to send output data.

Communication with a Sample Time Transition

For each communication channel you can specify the step size as an integer multiple of the basic real-time step size. The step size defines when data exchange on this communication channel takes place.

The step size of a communication channel must be an integer multiple of the step sizes of the connected source and destination CPUs.

Standard operation and downsampling

If both CPUs have identical step size multiples and you want to transfer data at a specific sample time (this corresponds to the behavior of a direct connection or a Zero Order Hold block), you have to enter identical numbers for the source and the destination step sizes (enter 1 for a direct connection). See also Example of Standard Operation of an IPC Block on page 139 and Example of Downsampling via an IPC Block on page 139.
Distributing the Model for MP Systems

**Different CPU step sizes**  
You can implement model parts with different sample times on various CPUs so that each CPU runs only one timer task and the single timer task mode is sufficient. As a result, you do not need the multiple timer task mode as often as you do with single-processor systems. The major benefit is that you do not need the Unit Delay block for the transition from the slow to the fast sampling rate, which avoids time delays. In this case, the source and destination step sizes are different. See also *How to Achieve Data Consistency Between Timer Tasks* on page 70 and *Example of an IPC Block with Different CPU Step Sizes* on page 142.

**Buffering and unbuffering**  
You can also use the IPC block to achieve data buffering or unbuffering. The buffering or unbuffering mode is selected automatically for an IPC connection as follows:

- **Step size of source CPU < step size of destination CPU**
  
The IPC connection is operated in buffering mode. For example, if the step size multiples are 1 for the source CPU and 10 for the destination CPU, and the number of buffered samples is 10, a vector of 10 input values is collected and passed to the destination CPU at every 10th simulation step.

- **Step size of source CPU > step size of destination CPU**
  
The IPC connection is operated in unbuffering mode. For example, if the step size multiples are 10 for the source CPU and 1 for the destination CPU, and the number of buffered samples is 10, a vector of 10 values is passed from the source to the destination CPU at every 10th simulation step. In each basic simulation step, the destination CPU uses one of the ten values as the model input.

You can use the **Buffered samples** edit field of the Communication Channel Setup dialog to define how many signal samples are to be buffered or unbuffered on an IPC connection.

Make sure that you transfer signals of the double data type and use the swinging buffer protocol. The IPC block does not support other data types or the virtual shared memory protocol if operated in the buffering or unbuffering mode.
See also Example of Buffering/Unbuffering via an IPC Block on page 142.

**Example of Standard Operation of an IPC Block**

The following illustration shows the behavior of an IPC channel for standard operation:
- Step size multiple of source CPU = 1
- Step size multiple of destination CPU = 1
- Step size multiple of channel for source CPU = 1
- Step size multiple of channel for destination CPU = 1
- Buffered samples = 1

**Example of Downsampling via an IPC Block**

The following illustration shows the behavior of an IPC channel for downsampling:
- Step size multiple of source CPU = 1
- Step size multiple of destination CPU = 1
- Step size multiple of channel for source CPU = 4
- Step size multiple of channel for destination CPU = 4
- Buffered samples = 1
Example application with downsampling

Suppose you want to operate the two CPUs of a multiprocessor system with the same step sizes (100 µs, the same as the base step size). There are two signals that you want to transfer from CPU master to CPU slave – one signal has to be transferred every 100 µs, the other at least every 500 µs. The following illustration shows a simplified model (the two signals have been replaced by a Sine Wave block) The different colors correspond to the sample times:

Because you want to operate the two IPCs at different sample times, you have to use two separate communication channels. Their settings are as follows:

- **master to slave on channel 0 (IPC1)** has to operate at 100 µs. Because this is the base step size, you have to enter a step size multiple of 1 for the communication channel.
- **master to slave on channel 1 (IPC2)** has to operate at 500 µs. This is five times the base step size, so you have to enter a step size multiple of 5 for the communication channel.
The following illustration shows the Communication Channel Setup dialogs for both IPC blocks:

IPC1 uses the step size multiple 1

IPC2 uses the step size multiple 5

This illustration shows the resulting signal plots on the CPU slave for both signals, along with the turnaround time, which shows peaks whenever the slow IPC transfers the data:
Example of an IPC Block with Different CPU Step Sizes

The following illustrations show the behavior of an IPC channel for CPUs operated with different step sizes.

**Source CPU faster than destination CPU**
- Step size multiple of source CPU = 1
- Step size multiple of destination CPU = 4
- Step size multiple of channel for source CPU = 1
- Step size multiple of channel for destination CPU = 4
- Buffered samples = 1

**Source CPU slower than destination CPU**
- Step size multiple of source CPU = 4
- Step size multiple of destination CPU = 1
- Step size multiple of channel for source CPU = 4
- Step size multiple of channel for destination CPU = 1
- Buffered samples = 1

Example of Buffering/Unbuffering via an IPC Block

The following illustrations show the behavior of an IPC channel for CPUs operated with different step sizes and the IPC channel operated in the buffer/unbuffer mode.

**Source CPU faster than destination CPU and IPC block in buffer mode**
- Step size multiple of source CPU = 1
- Step size multiple of destination CPU = 4
- Step size multiple of channel for source CPU = 1
- Step size multiple of channel for destination CPU = 4
- Buffered samples = 4
These settings correspond to the buffering operation of IPC1 in the FFT example shown below:

**Source CPU slower than destination CPU and IPC block in unbuffer mode**
- Step size multiple of source CPU = 4
- Step size multiple of destination CPU = 1
- Step size multiple of channel for source CPU = 4
- Step size multiple of channel for destination CPU = 1
- Buffered samples = 4

These settings correspond to the unbuffering operation of IPC2 in the FFT example shown below:

**Example application using the buffered and unbuffered modes**
Consider the FFT example in *Example Applications Requiring the Multiple Timer Task Mode* on page 69. The multiprocessor solution to the FFT problem is to assign the fast sampling to one CPU and the slow FFT to another one.
To run an FFT analysis, you have to assemble a specific number of input samples to a vector before you can process it. For example, if you need a vector of 1024 elements, you have to take 1024 samples (one at a time) and place them in an appropriate buffer. After the 1024th sample the vector is complete and can be processed by the FFT analysis. The following illustration shows the model for a multiprocessor system:

In order to achieve the desired data flow, you have to configure the model as follows:

- The base step size of the model has to correspond to your requirements, for example, 100 µs.
- The step size multiple of CPU master has to be 1 because it calculates the part related to the time domain (data acquisition or, in this case, signal generation via the Sine Wave block).
- The step size multiple of CPU slave has to be 1024 because you want it to collect the data of four sampling steps for each frequency domain computation.
- For the same reason, you have to configure both communication channels with 1024 buffered samples (depending on the source and destination step sizes of a communication channel, it automatically operates in either buffer or unbuffer mode).
- You also have to enter the source and destination step sizes for a communication channel according to the step sizes of its source and destination CPUs. This means that:
  - master to slave on channel 0 (IPC1) has a source step size multiple of 1 and a destination step size multiple of 1024,
  - slave to master on channel 1 (IPC2) has a source step size multiple of 1024 and a destination step size multiple of 1.
How to Define a Default CPU

You can use the Default CPU block to assign any stand-alone part of your model to a default CPU. A model part is stand-alone if it is left without a connection to an IPC block.

If there are stand-alone parts in your model that are not assigned to a default CPU, RTI-MP assigns them to the master CPU by default.

To define a default CPU for unconnected model parts

1. To assign unconnected model parts to a default CPU you have to use the Default CPU block. This block is located in the RTI-MP Blocks library.
   - To affect all stand-alone blocks of the entire model, including the subsystems, place a single Default CPU block on the root level.
To define a separate default CPU for a specific subsystem and all of its subordinate subsystems, place an additional Default CPU block inside that subsystem. You can use one Default CPU block per subsystem. All subsystems holding a Default CPU block have to be placed on the root level.

2 Open the Block Parameters dialog and specify the desired CPU.

If you add a new CPU to the model, you also need to configure the corresponding CPU's page of the Multiprocessor Setup dialog (see How to Configure an RTI-MP Model on page 120).
Interprocessor Interrupts

Two types of tasks exist in real-time applications: timer tasks and tasks driven by interrupt blocks. For dSPACE multiprocessor systems, you usually do not need to change anything for the timer tasks. However, tasks driven by interrupt blocks require a multiprocessor-specific treatment as described here.

There are a few RTI features that are not supported by RTI-MP. For details, see Limitations with RTI-MP on page 279.

Setting up interprocessor interrupts

Implement interprocessor interrupts to achieve synchronized tasks across several CPUs. Refer to How to Implement Interprocessor Interrupts on page 147.

Configuring priorities

Before building a real-time application you have to specify the execution order of the different tasks via priorities. Refer to How to Change the Task Priorities in RTI-MP Models on page 149.

How to Implement Interprocessor Interrupts

For all hardware, software or timer interrupts that are transferred from one CPU to another, you have to insert an Interprocessor Interrupt block (IPI block) into the connection lines between the interrupt block and the corresponding function-call subsystem.

The IPI block is not required for local interrupts, i.e., if the interrupt source and destination are located on the same CPU.
Consider the throttle control example (see Throttle Control Example on page 20). Suppose you want the emergency stop button (simulated by a step on the slave processor) to trigger the corresponding task on the master board. The necessary configuration is shown in the illustration below (the actual control loop is left out for clarity):

To set up a connection for interprocessor interrupts

1. Add and configure a Hardware, Software or Timer Interrupt block as usual (see How to Use Hardware Interrupts on page 87 and How to Use Software Interrupts on page 89).
   The block must be on the root level of the model.

2. Add an IPI block from the RTI-MP Blocks library to the root level of your model and connect it directly to the interrupt source.

3. Add a function-call subsystem from RTI's TaskLib to the root level of your model and connect it directly to the IPI block.
   No block other than a single IPI block is allowed between the interrupt source and function-call subsystem. You can drive only one subsystem with a single interrupt.

4. Open the function-call subsystem, and add the desired part of your model. If you want to assign different parts of this task to different CPUs, you can use IPC blocks and Default CPU blocks as though the function-call subsystem was the root level of the model.
Open the Block Parameters dialog of the IPI block. Specify the **Source CPU** and all Target CPUs involved in the execution of this task. You can find the involved Target CPUs by analyzing the IPC blocks and/or Default CPU blocks that the function-call subsystem might contain. If there are too few Target CPU edit fields you can expand the dialog by means of the **Maximum number of Target CPUs** drop-down list.

The following illustration shows an example:

![Illustration of Block Parameters Dialog](image)

### How to Change the Task Priorities in RTI-MP Models

If your model contains one or more interrupt-block-driven tasks, you have to specify priorities for each of them. If a task belongs to more than one CPU it can have different priorities on the various CPUs.

By choosing illogical priorities for the tasks of your model you might cause overrun or even deadlock.
To avoid overruns and deadlocks due to illogical priorities, it is important to assign the same priority order to tasks that are calculated on multiple CPUs. The following illustration shows an example:

You have to specify the task configuration individually for each CPU. The handling of the settings is identical to that of single-processor systems (see How to Change the Task Priorities on page 96). However, you cannot open them via the Simulation Parameters dialog because this dialog is not valid for real-time simulation on multiprocessor systems. Instead, follow the instructions shown below:

To open the RTI Task Configuration dialog for an MP system

1. Open the Multiprocessor Setup dialog.
2. Change to the individual CPU’s page, and open the RTI Task Configuration dialog for the current CPU.
3. Arrange the priorities according to your requirements, and close the RTI Task Configuration dialog.
4. Repeat steps 2 and 3 for each CPU included in your model.
Model Distribution Guidelines

The execution time of a multiprocessor application generally does not decrease linearly with the number of CPUs involved. This is due to the time overhead needed for interprocessor communication and to the delay that might result if you use the Synchronized Swinging Buffer protocol. Therefore, the execution time of a multiprocessor application (and thus its efficiency) depends highly on the structure of the application, meaning the amount and type of interprocessor communication used. The following guidelines show how to structure your model to achieve efficient multiprocessor application.

■ To achieve an efficiently parallelized execution, you should keep the amount of interprocessor communication as low as possible.

■ To avoid possible waiting times due to interprocessor communication, you might want to use the Unsynchronized Swinging Buffer protocol or the Virtual Shared Memory protocol instead of the Synchronized Swinging Buffer protocol (see Swinging Buffer Protocol on page 125 and Virtual Shared Memory Protocol on page 128). However, these protocols do not guarantee synchronization between sender and receiver, and the Virtual Shared Memory protocol does not even guarantee data consistency.

■ You should avoid direct feedthrough terms (such as IPC block – Gain block – IPC block) if you operate the corresponding communication channels with the Synchronized Swinging Buffer protocol. Otherwise, the involved CPUs might have to wait for each other due to a sequential working order. Direct feedthrough might even lead to deadlock situations. Refer to Model Evaluation on page 133 for details on the timing of the Synchronized Swinging Buffer protocol.

■ You should distribute the calculations equally over all CPUs.
You can free up execution time of the real-time application by assigning model parts (including IPC blocks) to the background task if these model parts are non-critical with regard to time. See Assigning Parts of the Model to the Background Task on page 93 for details.
Inserting Custom C Code

RTI is designed to provide automatic handling of dSPACE I/O board capabilities from a Simulink model. However, some specific hardware features are not supported by RTI. In this case, you can handle the appropriate hardware feature via a C-coded S-function or via RTI’s User-Code.

S-functions are a feature of Simulink that allow you to implement user-defined functionality. User-Code is a similar feature, but is provided by RTI. You can perform the following:

- Add customized blocks to a Simulink model
- Incorporate existing C code into a Simulink model
- Make dSPACE hardware accessible by implementing hardware access
Consider the throttle control example (see Throttle Control Example on page 20) and suppose you want to include a checking routine that issues a message if the throttle valve position exceeds a certain value. It is no problem to implement the checking routine into the Simulink model. However, no Simulink block provides message output from a real-time application. Thus, you have to use custom C code to achieve the desired behavior.

If you have a modular system containing a DS4201 Prototyping Board, you have to use an S-function or User-Code to access the custom I/O hardware capabilities of this board.

**Implementing**
You can implement your custom C code as S-functions. Refer to Implementing S-Functions on page 156.

You can implement your custom C code as User-Code. Refer to Implementing User-Code on page 164.

**S-function versus User-Code**
S-functions and User-Code have certain advantages. Refer to Comparing S-Functions and User-Code on page 174.

**Example**
The various aspects of the custom C code are explained via a custom C code problem for the level control example.

- There is a flow chart for this problem. Refer to Flow Chart for the Custom C Code of the Throttle Control Example on page 178.
- The solution is available as an S-function. Refer to C Code for the Throttle Control Example's S-Function on page 179.
- The solution is available as User-Code. Refer to C Code for the Throttle Control Example's User-Code on page 182.

**Accessing variables**
You can make the internal variables of the custom C code accessible to ControlDesk. Refer to How to Access Custom Variables via ControlDesk on page 184.

**Handling errors**
You can implement user-specific error handling in your custom C code. Refer to How to Handle Application-Specific Errors on page 186.
Slave applications  Some dSPACE boards have an I/O slave processor, and you might want to use it to run a specific slave application, for example, 4-channel square wave generation. Refer to Using Slave Applications on page 188.

Related Topics

Limitations  S-functions and User-Code have certain restrictions. Refer to Limitations for Custom C Code on page 280.
Implementing S-Functions

S-functions can be included in timer tasks or tasks driven by an interrupt block because you can place Simulink’s S-Function blocks anywhere in your model. In conjunction with the dSPACE real-time simulation you can only use C-coded S-functions.

Basics
For basic information on S-functions, refer to Basics of S-Functions on page 156.

Handcrafting
You can code the S-function methods in C, following the guidelines given in Handcrafting the C-Coded S-Function on page 158.

Related Topics

Useful examples
You can find useful examples of C-coded S-functions in Simulink’s S-functions demos – C-file S-functions library. You can use these demos as starting-points for your own S-functions. However, make sure that you do not use S-function demos for other languages than C because RTI supports only C-coded S-functions. For further information, refer to Writing S-Functions by The MathWorks. Details on dSPACE C functions for your board can be found in the corresponding RTLib Reference of the dSPACE HelpDesk.

Limitations
S-functions and User-Code have certain restrictions. Refer to Limitations for Custom C Code on page 280.

S-function techniques
You can find information on certain standard S-function techniques at http://www.dspace.de/goto?S-function_techniques.

Basics of S-Functions

S-functions (Simulink functions) provide a computer language description for Simulink’s S-Function blocks. S-functions are written in a high-level language such as M, C or Fortran.
If you want to use an S-function with a dSPACE system, it must be written in C. Real-Time Workshop and RTI do not support M, C++, Fortran, or Ada S-functions.

S-functions can hold continuous, discrete and hybrid systems and interact with the equation solvers of MATLAB and Simulink.

**Types of S-functions**

- **Noninlined S-functions** This type of S-function (also called C-MEX S-function) is used by Simulink and Real-Time Workshop, which call the various S-function methods during the model execution. Both Simulink and Real-Time Workshop use the same S-function code. This guide deals exclusively with this type of S-function.

- **Wrapper and fully inlined S-functions** These types of S-functions make use of the Target Language Compiler code generation tool. Fully inlined S-functions must be implemented separately for Simulink (C-MEX S-function) and Real-Time Workshop (TLC file). These types of S-functions are not described in this guide. For information, refer to the documentation of The MathWorks.

**S-function templates**

Since all S-functions must have a particular set of methods, you should always start your programming with an S-function template.

- If you want to create a new S-function (such as in the level control example), take the S-function template `sfuntmpl_basic.c` file from the `<MATLAB_ROOT>\simulink\src\` folder.

- If you want to modify an RTI-specific S-function, take the desired original S-function C file from the `%DSPACE_ROOT%\MATLAB\RTI<XXXX>\SFCn\` folder.

The original S-functions are restored and updated during any new installation. So before you modify any template, you should copy it to the working folder that holds your model and rename the template. It is then available for all models located in that folder.
S-function time overhead

Each S-function used in the Simulink model introduces an execution time overhead as listed in the table below. Level 1 S-functions always have a longer time overhead than level 2 S-functions.

<table>
<thead>
<tr>
<th>Board</th>
<th>Level 1</th>
<th>Level 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS1005 @ 480 MHz</td>
<td>0.5 µs</td>
<td>0.1 µs</td>
</tr>
<tr>
<td>DS1006 @ 2.2 GHz</td>
<td>0.097 µs</td>
<td>0.014 µs</td>
</tr>
<tr>
<td>DS1103 @ 333 MHz</td>
<td>1.2 µs</td>
<td>0.2 µs</td>
</tr>
<tr>
<td>DS1104 @ 250 MHz</td>
<td>2.5 µs</td>
<td>0.4 µs</td>
</tr>
<tr>
<td>DS1401 @ 300 MHz</td>
<td>2.3 µs</td>
<td>0.3 µs</td>
</tr>
</tbody>
</table>

Handcrafting the C-Coded S-Function

If you require user-specific functionality that is not based on an S-function predefined by RTI, you have to create it from scratch by using Simulink’s template `sfuntmpl_basic.c`. For example, this is the case for the message output of the throttle control example.

Make sure to follow the guidelines given here to avoid common difficulties when implementing S-functions.

Methods of an S-function

Each of the various C methods in an S-function has a specific purpose, which also qualifies it for specific dSPACE function calls. Refer to *Purpose of the C Methods* on page 159.

Real-time and Simulink simulations

You can write S-functions that use code variants for real-time and Simulink simulations, which is necessary if you want to use functions that are not available in one of the two environments. Refer to *S-Functions for Real-Time and Simulink Simulations* on page 160.

Accessing dSPACE I/O

Using the dSPACE real-time hardware from within C-coded S-functions lets you perform system initialization and access the I/O features. Refer to *How to Access dSPACE I/O Devices in S-Functions* on page 161.

Modular hardware

When accessing I/O boards via RTLib functions, you must always provide the PHS-bus base address as a parameter. Refer to *Accessing Modular Hardware in S-Functions* on page 162.
Related Topics

Handling errors
You might want to generate user-specific error messages within an S-function. Refer to How to Handle Application-Specific Errors on page 186.

RTLib functions
Details on the dSPACE C functions for your board can be found in the corresponding RTLib Reference available in the dSPACE HelpDesk.

S-functions guide
For further information, refer to External Interfaces/API and Writing S-Functions by The MathWorks.

Purpose of the C Methods

An S-function consists of a fixed set of mdl<...>() methods that are called by the mdl<...>() model functions, which Real-Time Workshop generates from the overall model. The set of mdl<...>() methods in an S-function can be considered as a model within the model.

mdlStart() Contains the initialization code of the S-function, which is executed once when simulation is started. You can use it for I/O initialization, for example dsXYZ_init() or dsXYZ_range_set().

mdlInitializeConditions() Sets the initial values of the state variables of the S-function. It is executed once when the simulation is started. If the S-Function block is in an enabled subsystem that is configured as “reset states when enabled”, it is also executed whenever the subsystem becomes enabled. Therefore, this method is generally not suitable for I/O initialization code.

mdlInitializeSizes() Defines the size characteristics of the S-function, such as the number of inputs, outputs, states and sample times. Simulink reads the characteristics to configure the S-Function block.

mdlInitializeSampleTimes() Defines the sample times for the S-function.

mdlOutputs() Contains the code for calculating the block outputs of the S-function. You can use it for read/write access of I/O devices.

mdlUpdate() Contains the algebraic code for calculating the discrete states of the S-function.
**mdlDerivatives()**  Contains the code for calculating the derivatives of time-continuous state variables within the integration routine.

**mdlTerminate()**  Contains the termination code of the S-function, which is executed once when simulation is stopped. You can use it for writing the termination values to the output devices.

If an I/O device is initialized more than once, for example, if you place the initialization in `mdlInitializeConditions()` by mistake, the behavior of the device can become unpredictable. Therefore, use `mdlStart()` for initializations that must occur only during simulation start.

**S-Functions for Real-Time and Simulink Simulations**

Since S-functions are a feature of Simulink, you can use them not only for real-time simulation, but also for standard Simulink simulation. Even though the majority of the code of an S-function is valid for both real-time and Simulink simulations, you usually need to conditionally include or exclude specific code segments for one of the two purposes. As an example, consider dSPACE I/O statements that do not make sense for Simulink simulation. On the host PC system there is no dSPACE hardware available and dSPACE I/O functions are not valid.

You can achieve this conditional inclusion or exclusion of a code segment via the `MATLAB_MEX_FILE` C preprocessor variable, which is defined during the MEX DLL file generation and not defined when real-time code is generated. Use the syntax shown below:

```c
#ifdef MATLAB_MEX_FILE
    /* Code only used for Simulink simulation that */
    /* you want to exclude from the real-time object file, */
    /* for example, an output on the host computer’s screen. */
#else
    /* Code only used for real-time simulation that */
    /* you want to exclude from the Simulink S-function, */
    /* for example, dSPACE-specific I/O code. */
    /* Instead of the #ifdef ... #else construct you can also*/
    /* use the line ifndef MATLAB_MEX_FILE; for example, */
    /* if there is no need for Simulink-only code. */
#endif
```
Any line of code that is not enclosed in such an `#ifdef` or `#ifndef` construct is included in both real-time code generation and Simulink simulation.

Consider the throttle control example: The statements needed for the real-time simulation that send a message to the Message Module (i.e. ControlDesk’s Log Viewer) are functional only for the real-time simulation. Therefore you have to tag them with the `#ifndef MATLAB_MEX_FILE` statement. On the other hand, you have to conditionally include a statement that sends the message to the MATLAB Command Window if the model is simulated in Simulink. Thus, you have to enclose it within an `#ifdef MATLAB_MEX_FILE` construct. The corresponding code segment is shown below:

```c
static void mdlOutputs(SimStruct *S, int_T tid)
{
    /* Get pointer for input port one.
       Note: "uPtrs" is a vector of void pointers. It is
       absolutely crucial to cast the indexed element from void
       pointer to the proper type of pointer. Only after the
       cast is it safe to get the value via indirection. */
    InputPtrsType uPtrs = (InputPtrsType) ssGetInputPortSignalPtrs(S, 0);
    boolean_T Limit = *((boolean_T *) uPtrs[0]);
    /* Check if limit is reached. */
    if (Limit)
    {
        #ifdef MATLAB_MEX_FILE
        /* Warning for Simulink simulation. */
        mexWarnMsgTxt("Limit reached!");
        #else
        /* Warning for real-time simulation. */
        msg_warning_set(MSG_SM_USER, 1, "Limit reached!");
        #endif
    }
}
```

**How to Access dSPACE I/O Devices in S-Functions**

Using the dSPACE real-time hardware from within C-coded S-functions lets you perform system initialization and access the I/O features.
To use RTLib C functions and macros for I/O access

1. To access dSPACE I/O via an S-function, you must include the following lines in the S-function header:

   ```c
   ifndef MATLAB_MEX_FILE
   # include <brtenv.h> /* basic real-time environment */
   # include <dsXXXX.h> /* dsXXXX Board, e.g. ds2001.h */
   #endif
   ```

2. Make sure to also enclose all subsequent RTLib statements such as I/O access within the `ifndef MATLAB_MEX_FILE ... #endif` statements.

Since the `MATLAB_MEX_FILE` macro is only defined if the S-function is compiled for a Simulink simulation (i.e., when building a MEX DLL file), the dSPACE-specific statements are not executed for Simulink simulations but only for real-time simulation.

If you want to provide extra code for Simulink simulations that is not relevant to dSPACE real-time simulations, for example, screen output for the MATLAB Command Window, you can use the `#ifdef MATLAB_MEX_FILE ... #endif` statements.

Accessing Modular Hardware in S-Functions

For modular dSPACE systems there is a distinct relationship between the PHS-bus address of an I/O board and its board number as shown in RTI. Refer to PHS-Bus Address and Board Number on page 43.

When accessing I/O boards via RTLib functions, you must always provide the PHS-bus base address as a parameter. You can obtain this address via the `DSXXXX_n_BASE` macros that are made available via the hardware-specific `brtenv.h` header.

Suppose you want to stop channel 1 of the timer that is located on the second DS4001 board of your system. In this case, specify the following include statement in the header of your S-function:

```c
#include <brtenv.h>
#include <ds4001.h>
```
You can then use the DS4001_2_BASE macro to retrieve the board address needed for further RTLib I/O functions, in this case:

ds4001_timer_stop(DS4001_2_BASE, 1);

The ds4001.h header file contains the board-specific macros, for example, ds4001_timer_stop().

A more general approach is to pass the board number to the S-function as a parameter, which makes the S-function more flexible towards changing the dSPACE system. As a result, you do not need to modify the S-function when adding further I/O boards. In this case, use the following macro:

PHS_BOARD_BASE_GET(board_type, board_number)

where

board_type = PHS board type macro, for example, PHS_BT_4001 for a DS4001 board

board_number = board number to distinguish between boards of the same type. If board_number is passed to the S-function by argument, you can access the argument via the ssGetSFcnParam statement.
Implementing User-Code

With User-Code, RTI offers a set of adaptable C functions that allow you to customize the initialization segment, the evaluation sequence of the fastest timer task, the background task, and the termination segment. User-Code can be accessed in the Simulink model by means of Simulink's Data Store Memory blocks.

Transferring data

You need to prepare the communication between the model and User-Code. Refer to How to Use User-Code in Simulink Models on page 164.

Generating a template

If not already done, you must generate a template for the User-Code. Refer to How to Generate a Template for the User-Code on page 167.

Handcrafting

You can code the desired functions in C, following the guidelines given in Handcrafting the User-Code on page 168.

Related Topics

Limitations

S-functions and User-Code have certain restrictions. Refer to Limitations for Custom C Code on page 280.

How to Use User-Code in Simulink Models

Consider the throttle control example (see Throttle Control Example on page 20) and suppose you want to receive a message if the throttle valve position exceeds 50°. The Boolean value (whether the limit is reached or not) can be calculated in the Simulink model. You can then let the User-Code output the appropriate message.
To transfer the Boolean value from the Simulink model to the User-Code you have to change your model as illustrated below:

As you can see from this model, the interface between Simulink and User-Code is implemented via Simulink’s Data Store blocks:

- Each Data Store Memory block defines and initializes a single communication variable.
- The Data Store Read block represents the transition from the User-Code to the Simulink model.
The Data Store Write block denotes the transition from the Simulink model to the User-Code.

Since these Data Store blocks do not have any function on their own (apart from the communication), you have to hand-code the desired function in C.

If there are any Data Store Memory blocks in your Simulink model, you get a separate Data Stores group in the Variable Browser of ControlDesk. This group allows you to access the communication variables of the User-Code.

To include User-Code in a Simulink model

1. For each communication variable needed, place a separate Data Store Memory block from the Simulink library into your model.

2. For each communication variable, open the Block Parameters dialog of the corresponding Data Store Memory block and specify the following settings.
   - Enter the name of the communication variable (for example, Limit) as Data store name.
   - Enter the desired initial value and – via the vector width – its size.
   - Set the RTW Storage Class to “ExportedGlobal”.

3. To transfer data from the User-Code to the Simulink model, place a Data Store Read block from the Simulink library in your model. To transfer data from the Simulink model to the User-Code, use the Data Store Write block instead. Again, you need separate blocks for each communication variable you want to access.

4. For each block, open the Block Parameters dialog and specify the data store name of the communication variable you want to access: for example, Limit. This name must match the name you gave to the corresponding Data Store Memory block. You can also specify a Sample time for the communication variable.

5. The User-Code always belongs to the fastest timer task and, therefore, has the same priority. If the Simulink part of a communication variable does not belong to the timer task driven...
with the base sample time, it might belong to a different task. If this is the case and if the variable is a vector, you have to avoid data inconsistency:

- If the User-Code has the higher priority, you have to include the Uninterruptable Buffer in the Simulink model as described in Preparing a Function-Call Subsystem on page 78.
- If the User-Code has the lower priority this is not necessary, but you have to take precautions inside the User-Code. You find the necessary instructions in How to Handcraft the User-Code on page 169.

**How to Generate a Template for the User-Code**

If you have not built real-time code for your Simulink model yet, there is no User-Code template C file yet. However, if you want to program the User-Code, you need the User-Code template C file for the model. To generate this file, perform the following steps:

**To get a User-Code template**

1. Change to the working folder that holds your model.
2. Start a build process for the Simulink model you want to work on (see How to Start the Build and Download on page 248).

Once you have built your model you will find the `<model>_usr.c` file in its working folder. This template contains a set of empty C functions into which you can incorporate your custom C code.

RTI generates a new User-Code template C file only if there is no `<model>_usr.c` file in the working path of the model yet. Thus, if you already added user-specific code to the User-Code file of the model, RTI does not replace this file.

**RTI-MP** For RTI-MP models there are several User-Code files – one for each CPU of the model. These User-Code files are named according to the applications of the different CPUs: `<submodel>_usr.c`. 

---

**RTI and RTI-MP Implementation Guide**

March 2005
Handcrafting the User-Code

The User-Code template file `<submodel>_usr.c` contains a set of empty C functions. Each C function has a specific purpose. You have to enter your user-specific C code in these C functions according to their purposes.

To transfer data from the User-Code to the Simulink model (for example, from an input device) you have to use the function `usr_input()`. If you want to transfer data from the Simulink model to the User-Code (for example, to send it to an output device) you have to use the function `usr_output()`. The additional User-Code function `usr_sample_input()` can be used to start the conversion process for an ADC. This process is then finished before the `usr_input()` function starts, which reduces delays in execution.

**Communication with Simulink**

The User-Code and Simulink can communicate by means of a global structure variable. Refer to *Communication Between the User-Code and Simulink* on page 169.

**Handcrafting the User-Code**

Implement the desired functionality in the User-Code file. Refer to *How to Handcraft the User-Code* on page 169.

**Setting up interprocessor communication**

With RTI-MP, data is limited to the User-Code of a single CPU. To transfer data from the User-Code of one CPU to the User-Code of another, you can implement user-defined interprocessor communication on C level. Refer to *User-Code Interprocessor Communication* on page 171.

**Related Topics**

- **Board-specific RTLib functions**
  Details on the dSPACE C functions for your board can be found in the corresponding *RTLib Reference* available in the dSPACE HelpDesk.
Communication Between the User-Code and Simulink

The Data Store blocks of a Simulink model whose RTW Storage Class is set to “ExportedGlobal” are available via the names specified as Data store name, for example, Limit. If such a communication variable is a data vector, you can access its individual elements by means of an index; for example, Limit[0].

If you use such vectorized communication variables, the corresponding index in the User-Code runs from 0 to n-1 if n is the number of vector elements. Do not use any other indices, because this corrupts the memory of the real-time processor.

How to Handcraft the User-Code

To handcraft the User-Code

1 Fill the empty functions of the <model>_usr.c User-Code template C file with your user-specific C code. Place the code statements in the appropriate functions according to their purposes as specified below:

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Target Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization, e.g., write initialization values to output devices</td>
<td>usr_initialize()</td>
</tr>
<tr>
<td>Start conversion, e.g., for A/D devices</td>
<td>usr_sample_input()</td>
</tr>
<tr>
<td>Read access, e.g., for input devices</td>
<td>usr_input()</td>
</tr>
<tr>
<td>Write access, e.g., for output devices</td>
<td>usr_output()</td>
</tr>
<tr>
<td>Shut down, e.g., write termination values to output devices</td>
<td>usr_terminate()</td>
</tr>
<tr>
<td>Background task</td>
<td>usr_background()</td>
</tr>
</tbody>
</table>

2 Where required, implement the communication with the Simulink model:
   - You can access the Data Store Read and Data Store Write blocks of the Simulink model via the Data store names specified in the blocks.
Inserting Custom C Code

- Do not exceed the defined index if you use vectorized variables.
- If the Simulink part of a communication variable does not belong to the timer task driven with the base sample time, it might belong to a different task. If this is the case and if the variable is a vector, you have to avoid data inconsistency:
  - If the User-Code has the lower priority, you have to enclose the critical code segments that access the communication variables with the `global_disable()` and `global_enable()` statements. Refer to the `RTLib Reference` (available in the dSPACE HelpDesk) for further information on these functions.
  - If the User-Code has the higher priority, this is not necessary. Take precautions in the Simulink model as described in *How to Use User-Code in Simulink Models* on page 164.

Consider the throttle control example. The following statements form the basic function that outputs the message to the Message Module if the throttle valve position exceeds 50°.

```c
__INLINE void usr_output(void)
{
    /* check if limit is reached */
    if (Limit!=0.0)
    {
        /* warning for real-time simulation */
        msg_warning_set(MSG_SM_USER, 1,
                        "Throttle Control Example: Limit reached!");
    }
}
```

To keep this code segment simple it does not contain administrative code (generating a single message output even if the throttle valve position remains above the limit). The complete source code is included in *C Code for the Throttle Control Example’s User-Code* on page 182.
Inserting Custom C Code

User-Code Interprocessor Communication

Between two processor boards of the same type, you can use as many user-defined data channels as are left unused by the Simulink model. Each GigaLink provides eight shared memory (SMEM) and eight swinging buffer (SBUF) channels in total. You can get the numbers of the GigaLink and channel via the function \texttt{rtimp\_get\_usr\_channel}. This function has the following syntax:
\begin{verbatim}
rtimp\_get\_usr\_channel(*<CPU>*\, <protocol>, <No>, &gl, &chan);
\end{verbatim}
The numbers of the free IPC channels (the GigaLink and channel numbers) are assigned dynamically, depending on the channels used by the Simulink model.

The table below lists the purposes of the terms:

\begin{center}
\begin{tabular}{|c|p{0.8\textwidth}|}
\hline
\textbf{Term} & \textbf{Purpose} \\
\hline
<CPU> & Specify the name of the CPU that you want to communicate with. \\
\hline
<protocol> & Specify the desired protocol. \\
IPC\_SBUF means the swinging buffer protocol, IPC\_VSM means the virtual shared memory protocol. \\
\hline
<No> & Specify the desired relative index number for the channel. For example, if you specify IPC\_VSM as the protocol and 1 as the relative index number, you get the numbers for the first free virtual shared memory channel, regardless of how many virtual shared memory channels are used by the IPC blocks of the Simulink model. \\
\hline
&gl & This refers to the \texttt{gl} variable that returns the number of the GigaLink to be used. \\
\hline
&chan & This refers to the \texttt{chan} variable that returns the number of the channel to be used. \\
\hline
\end{tabular}
\end{center}
Suppose you want to transfer the content of the `out_data` variable from the User-Code of the master to the `in_data` variable of the slave's User-Code. You want to use the first available swinging buffer channel. The following code excerpt shows how to initialize and access the required channel in the User-Code of the master board:

```c
int out_data = 0;
int *ch_ptr;
int gl;
int chan;
static void usr_initialize(void)
{
    /* get number of gigalink and channel */
    rtimp_get_usr_channel("slave",IPC_SBUF,1,&gl,&chan);
    /* pointer to the user channel */
    ch_ptr = dsgl_ptr_get(gl,chan);
}

__INLINE void usr_output(void)
{
    /* calculate output data*/
    out_data++;
    /* send output and switch buffer */
    *ch_ptr = out_data;
    dsgl_write_buffer_switch(gl,chan);
}
```

The corresponding code excerpt in the User-Code of the slave board looks like this:

```c
int in_data;
int *ch_ptr;
int gl;
int chan;
static void usr_initialize(void)
{
    /* get number of gigalink and channel */
    rtimp_get_usr_channel("master",IPC_SBUF,1,&gl,&chan);
    /* pointer to the user channel */
    ch_ptr = dsgl_ptr_get(gl,chan);
}
```
__INLINE void usr_output(void)
{
    /* switch buffer, if buffer has been updated */
    if (dsgl_read_buffer_is_updated(gl,chan)){
        dsgl_read_buffer_switch(gl,chan);
    }
    /* get input data */
    in_data = *ch_ptr;
}
Comparing S-Functions and User-Code

Both S-functions and User-Code have specific advantages and disadvantages, which are listed below. These can help you decide whether you want to use an S-function or User-Code.

<table>
<thead>
<tr>
<th></th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-function</td>
<td>Can be executed in any task of the real-time program.</td>
<td>No control over the time of initialization/execution in the overall execution order of the model. Source code not as easy as that of User-Code. Always executed with the fastest timer task.</td>
</tr>
<tr>
<td>User-Code</td>
<td>Precise control over the time of initialization/execution in the overall execution order of the model. Source code simpler than that of S-functions.</td>
<td></td>
</tr>
</tbody>
</table>

Comparing the execution order
Both User-Code and S-functions provide a set of C functions or methods, which are executed in a specific order during the overall model evaluation. Refer to Execution Order of S-Functions and User-Code on page 175.

Tips and tricks
Several tips might help you implement your custom C code. Refer to Tips and Tricks for Custom C Code on page 177.

Related Topics
Restrictions
S-functions and User-Code have certain restrictions. Refer to Limitations for Custom C Code on page 280.
Execution Order of S-Functions and User-Code

The following two flow charts summarize when the individual C functions or methods of User-Code and S-functions are executed in relation to the Simulink model.

- Certain RTI blocks are implemented as S-functions. They are executed like any other user-specific S-functions. The execution order of S-functions is demonstrated via the fictional Sfcn A and Sfcn B blocks. All other RTI blocks are explicitly shown in the flow chart.

- Wherever you find a box containing a number of flow chart blocks, the execution order for these cannot be predefined.
Execution order of the tasks of a model

This flow chart shows the execution order of the initializations and blocks that are contained in the timer tasks. Whenever all these tasks are finished, the processor executes the background task, which is shown in the next flow chart.

- **Sfcn A: mdlInitializeSizes()**
- **mdlInitializeSampleTimes()**
- **Sfcn B: mdlInitializeSizes()**
- **mdlInitializeSampleTimes()**
- **Sfcn A: mdlStart()**
- **Sfcn B: mdlStart()**
- **Sfcn A: mdlInitializeConditions()**
- **Sfcn B: mdlInitializeConditions()**
- **usr_initialize()**
- **RTI blocks: I/O device initialization**
- **RTI blocks: write initial values to output devices**
- **usr_sample_input()**
- **RTI blocks: sample & read inputs**
- **usr_input()**
- **usr_initialize()**
- **Model code from Simulink blocks**
- **Sfcn A: mdlOutputs()**
- **Sfcn B: mdlOutputs()**
- **RTI blocks: write to output devices**
- **usr_output()**
- **simState?**
  - **RUN**
  - **STOP**
- **Model termination code from Simulink blocks**
- **Sfcn A: mdlTerminate()**
- **Sfcn B: mdlTerminate()**
- **RTI blocks: write termination values to output devices**
- **usr_terminate()**
- **simState?**
  - **RUN**
  - **STOP**
Inserting Custom C Code

Execution order of the background task of a model

This flow chart shows the execution order of the blocks that are assigned to the background task. The background task is executed only if no other task (shown in the previous flow chart) needs to be executed.

Tips and Tricks for Custom C Code

The following tips might help you implement your custom C code.

Holding C files separately
If you want to hold your S-function C files in a separate folder, you have to specify this folder in the user makefile. Refer to User Makefile (USR.MK File) in the RTI and RTI-MP Implementation Reference.

Including files and libraries
If you want to include and use custom source files, include files, or libraries in S-functions or User-Code, you have to specify these files together with their paths in the user makefile. Refer to User Makefile (USR.MK File) in the RTI and RTI-MP Implementation Reference.

Rebuilding custom C code
You do not always need to start a complete build process after you modify certain parts of the custom C code. Instead, you can run the make utility by hand. Refer to How to Automate the Build Procedure on page 250.
Flow Chart for the Custom C Code of the Throttle Control Example

The following flow chart shows the behavior that has to be implemented in an S-function or User-Code to solve the required job for the throttle control example. The white blocks issue the actual warning if the throttle valve position exceeds the limit, the light-colored blocks ensure that the warning is issued only once, and the dark-colored block counts how often the warning was issued.
C Code for the Throttle Control Example’s S-Function

The following C source code forms the required S-function:

```c
/* FILE: issue_warning.c
DESCRIPTION:
Example S-function for the throttle control example from the RTI Implementation Guide. The S-function issues a warning if the input is logical true.
REMARKS:
This file is based on the Simulink S-function template sfuntmpl_basic.c and contains only the most common S-function methods. Comments on example-specific changes are included.
See <MATLAB_ROOT>/simulink/src/sfuntmpl_doc.c for more details.
Copyright (c) 1997-2005 dSPACE GmbH, GERMANY
*******************************************************************************/
#define S_FUNCTION_LEVEL 2
#define S_FUNCTION_NAME issue_warning
#include "simstruc.h"
/* When compiling for dSPACE target, include the brtenv.h file.*/
#ifndef MATLAB_MEX_FILE
# include <brtenv.h>
#endif
/* Declare global variables. These can be made available in ControlDesk by adding them to the <model>_usr.trc file.
* NOTE:
* Due to the use of global variables, this S-function is not reentrant */
int warning_flag = 0; /* Control issuing of warning */
int counter = 0; /* Counter for warnings */
/* Function: mdlInitializeSizes =========================== */
static void mdlInitializeSizes(SimStruct *S)
{
    /* This S-function has no parameters. */
    ssSetNumSFcnParams(S, 0); /* No. of expected parameters */
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) return;
    /* Set the number of states to zero. */
    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 0);
    /* Set one input port. */
    if (!ssSetNumInputPorts(S, 1)) return;
    /* Set width of input port to one. */
    ssSetInputPortWidth(S, 0, 1);
    /* Input is accessed in mdlOutputs() routine, therefore set direct feedthrough flag. */
    ssSetInputPortDirectFeedThrough(S, 0, 1);
    /* Set the input port data type to boolean. */
    ssSetInputPortDataType(S, 0, SS_BOOLEAN);
    /* We have no output ports, therefore we must not call ssSetOutputPortWidth() */
    if ((ssGetNumOutputPorts(S, 0)) return;
    /* This S-function has one sample time. */
    ssSetNumSampleTimes(S, 1);
    /* No work vectors and other special features are used. */
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
}
```
ssSetNumWork(S, 0);
ssSetNumModes(S, 0);
ssSetNumUnsampledZCs(S, 0);
/* No additional options are used. */
ssSetOptions(S, 0);
}

/* Function: mdlInitializeSampleTimes ========================= */
static void mdlInitializeSampleTimes(SimStruct *S)
{
/* Use inherited sample time. */
ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
/* Ensure that this S-function will not execute in intermediate integration steps. */
ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);
}

#define MDL_START /* Change to #undef to remove function */
#if defined(MDL_START)
/* Function: mdlStart =================================== */
static void mdlStart(SimStruct *S)
{
/* Reset control variables when switching RTI simulation state from STOP to RUN. */
warning_flag = 0;
counter = 0;
}
#endif /* MDL_START */

static void mdlOutputs(SimStruct *S, int_T tid)
{
/* Get pointer for input port one. 
   Note: "uPtrs" is a vector of void pointers. It is absolutely crucial to cast the indexed element from void pointer to the proper type of pointer. 
   Only after the cast is it safe to get the value via indirection. */
InputPtrsType uPtrs = (InputPtrsType) ssGetInputPortSignalPtrs(S, 0);
boolean_T Limit = *((boolean_T *) uPtrs[0]);
/* Check if limit is reached. */
if (Limit)
{
/* Check if warning has already been issued. */
if (!warning_flag)
{
/* Set flag to remember that warning has been issued. */
warning_flag = 1;
/* Increment the number of warnings. */
counter++;
#ifdef MATLAB_MEX_FILE
/* Warning for Simulink simulation. */
mxWarningMsgTxt("Throttle Control Example: Limit reached!");
#else
/* Warning for real-time simulation. */
msg_warning_set(MSG_SM_USER, 1, "Throttle Control Example: Limit reached!");
#endif
}
else
{
/* Reset the warning flag. */
warning_flag = 0;
}
}
/* Function: mdlTerminate ----------------------------------------*/
static void mdlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE /* Compile as a MEX file? */
#include "simulink.c" /* MEX file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration fcn */
#endif
C Code for the Throttle Control Example’s User-Code

The following C source code forms the required User-Code.

/**************************************************************************
Include file throttlecontroldemo_usr.c:
Definition of functions for user-defined initialization,
system I/O, and background process code.
RTI1401 5.0
<Date>
Copyright (c) 1997-2005 dSPACE GmbH, GERMANY
***************************************************************************/
/* =========================================================================
/* ===== Define file version macro. Never change this value. =============
/* =========================================================================
#define USER_CODE_FILE_VERSION 5
/* =========================================================================
/* Insert #include directives for header files here. */
#if defined(__INLINE)
# define __INLINE static inline
#else
# define __INLINE static
#endif
/* declare global variables */
int warning_flag = 0; /* to control issuing of warning */
int counter = 0; /* counter for warnings */
static void usr_initialize(void)
{
    /* reset warning flag if simulation starts (STOP -> RUN) */
    warning_flag = 0;
    /* reset counter if simulation starts */
    counter = 0;
}
__INLINE void usr_sample_input(void)
{
}
__INLINE void usr_input(void)
{
}
__INLINE void usr_output(void)
{
    /* check if limit is reached */
    if (Limit != 0.0)
    {
        /* check if warning has already been issued */
        if (warning_flag == 0)
        {
            /* Set flag to remember that warning has been issued. */
            warning_flag = 1;
            /* Increment the number of warnings. */
            counter++;
            /* warning for real-time simulation */
            msg_warning_set(MSG_SM_USER, 1,
                            "Throttle Control Example: Limit reached!");
        }
    }
}
```c
else
{
    /* reset warning flag */
    warning_flag = 0;
}

static void usr_terminate(void)
{
}

static void usr_background(void)
{
}

#undef __INLINE

****** [EOF] ****************************************************************/
```
How to Access Custom Variables via ControlDesk

Suppose you want to capture custom variables of an S-function or User-Code. By default, RTI does not include the internal variables of S-functions or User-Code in the variable description file, only the output of S-function blocks or the global Data Store Memory communication variables.

If you are using more than one instance of a specific S-function in your model, take care to avoid unexpected side-effects in your application that might occur when using global variables.

Accessing custom variables

It is possible to access variables of your S-function or User-Code:

To make custom variables accessible for ControlDesk

1. In the corresponding S-function or User-Code, declare the custom variables as global non-static variables or as global non-static pointer variables.

2. Add a user variable description file to the folder of the model (<model>_usr.trc for single-processor systems or <submodel>_usr.trc for multiprocessor systems).

3. List the desired custom variables in the user variable description file using the syntax of the variable description file (see Syntax of the TRC File in the ControlDesk Reference).

4. Build your real-time application and download it to the real-time hardware (see Building and Downloading the Model on page 241). During the build process, RTI adds the contents of the <model>_usr.trc file to the <model>.trc file.

5. In ControlDesk, load the corresponding system description file (<model>.sdf) to the specified dSPACE board. The Variable Browser displays the internal variables in an extra group called User Variables.
Consider the throttle control example (see page 20). Suppose you want to implement a counter in the user-defined code that counts how often a message is issued that warns that the throttle valve position has exceeded its limit. The corresponding code could read as follows:

```c
/* Declare global variables. These can be made available in ControlDesk by adding them to the <model>_usr.c file. */
int warning_flag = 0; /* Control issuing of warning */
int counter = 0; /* Counter for warnings */
```

The contents of `<model>_usr.trc` would read as follows:

```plaintext
-- ******** Warning control variables***************
warning_flag
{  
type: int(32)
   desc: "Control issuing of warning"
}
counter
{  
type: int(32)
   desc: "Counter for warnings"
}
```

As a result, ControlDesk’s Variable Browser would look like the following illustration:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Size</th>
<th>Type</th>
<th>Align</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>warning_flag</td>
<td>1x1</td>
<td>int32</td>
<td></td>
<td>Control issuing of warning</td>
</tr>
<tr>
<td>counter</td>
<td>1x1</td>
<td>int32</td>
<td></td>
<td>Counter for warnings</td>
</tr>
</tbody>
</table>
How to Handle Application-Specific Errors

If you want to generate messages in certain situations and display them via the Message Module of the host computer, you have to use special C-function calls. You can use these C-function calls in any S-function or User-Code. The following instructions summarize the necessary steps:

To send messages to the Message Module

1. Include the `#include <brtenv.h>` statement in the program header of your S-function or User-Code.
2. Use the access C functions according to the table shown below to send information, a warning or an error to the Message Module. An error message automatically terminates the simulation by switching the `simState` variable to STOP.

<table>
<thead>
<tr>
<th>Desired Operation</th>
<th>Message Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display information</td>
<td><code>msg_info_set()</code></td>
</tr>
<tr>
<td>Display warning</td>
<td><code>msg_warning_set()</code></td>
</tr>
<tr>
<td>Display error, terminate simulation</td>
<td><code>msg_error_set()</code></td>
</tr>
<tr>
<td>Return last error number</td>
<td><code>msg_last_error_number()</code></td>
</tr>
</tbody>
</table>

You have to use these access C functions together with the submodule identifier `MSG_SM_USER`.

For details on the Message Module, refer to Message Handling in the RTLib Reference that corresponds to your hardware.

Reporting errors

For reporting errors you can use the `ssSetErrorStatus()` function in your S-function or User-Code, which works for both Simulink and real-time simulation. If your S-function or User-Code reports an error with the `ssSetErrorStatus()` function, the message used with this function is displayed in ControlDesk’s Log Viewer, and RTI stops the application. Then the real-time application is terminated via the `exit()` function (see also Reaction to Run-Time Errors on page 208).
If you want to report a problem without terminating the real-time application, you cannot use the `ssSetErrorStatus()` function. Instead use the `msg_warning_set(MSG_SM_USER, 1, <...>)` or `msg_warning_printf(MSG_SM_USER, 1, <...>)` functions for the real-time application and the `ssWarning(S, <...>)` function for the Simulink simulation.

For details on `ssSetErrorStatus` and `ssWarning` see *Writing S-Functions* by The MathWorks.
Using Slave Applications

Some dSPACE boards have an I/O slave processor, and you might want to use it to run a specific slave application, for example, 4-channel square wave generation. This is possible via the dSPACE standard application 4df.

Examples

Consider the following examples:

- Example of a Slave Application as User-Code on page 188
- Example of a Slave Application as an S-Function on page 191

Some of the standard applications for the I/O slave processors are available as blocks from the corresponding RTI library.

Related Topics

For hardware-specific information on slave applications, refer to Loading Slave Applications in the DS2210 RTLib Reference and Slave-DSP Applications in the DS2201 RTLib Reference.

Example of a Slave Application as User-Code

Suppose you run a DS2201 board and want to use its slave processor to run a 4-channel square wave generation. Include the 4df.c file in the User-Code of your model via the

#include <%DSPACE_ROOT%\Slave\DS2201\4df.c> C include statement.
The corresponding Simulink model could look like this:

The DS2201-specific User-Code for this application could read:

```c
#include slaveapp_usr_ds2201_usr.c:
Definition of functions for user-defined initialization, system I/O, and background process code.
DESCRIPTION:
User Code file to perform a 2-channel frequency measurement and 2-channel signal generation based on the 2fd2df slave application (DS2201 version).
Copyright (c) 1997-2005 dSPACE GmbH, GERMANY
******************************************************************************
/* = Define file version macro. Never change this value. = */
#define USER_CODE_FILE_VERSION 5
/* Insert #include directives for header files here. */
#include <brtenv.h>
#include <ds2201\4DF.c>
#if defined(_INLINE)
#define __INLINE static inline
#else
#define __INLINE static
#endif
#include <ds2201\slaveapp\ds2201\2fd2df.c>
```
static void usr_initialize(void)
{
/* Initialize the DS2201 I/O board. */
ds2201_init(DS2201_1_BASE);
/* Load the slave application. */
load_slave_4df(DS2201_1_BASE);
/* Enable signal generation. */
ds2201_dtof_enable(DS2201_1_BASE, DS2201_DTOF_ENABLE);
}

__INLINE void usr_sample_input(void)
{
}

__INLINE void usr_input(void)
{
}
__INLINE void usr_output(void)
{
/* Write frequency value for channel 1. */
ds2201_dtof_4(DS2201_1_BASE, 1, freq_out[0]);
/* Write frequency value for channel 2. */
ds2201_dtof_4(DS2201_1_BASE, 2, freq_out[1]);
/* Write frequency value for channel 3. */
ds2201_dtof_4(DS2201_1_BASE, 3, freq_out[2]);
/* Write frequency value for channel 4. */
ds2201_dtof_4(DS2201_1_BASE, 4, freq_out[3]);
}

static void usr_terminate(void)
{
}

static void usr_background(void)
{
}

#undef __INLINE
/** [EOF] */

The following actions are carried out in this User-Code:

- The source code of the 4df slave application is included. This code also incorporates the necessary code for loading the application from the master CPU to the slave CPU, which makes it possible to load and start the slave application with the call `load_slave_4df()`.  
- Via the Data Store Write block, the frequency value of the generated square-wave signal is passed to the User-Code to let you control the frequency parameter from within the Simulink model. In the User-Code the corresponding variables are

  \[
  \begin{align*}
  \text{freq\_out}[0] \\
  \text{freq\_out}[1] \\
  \text{freq\_out}[2] \\
  \text{freq\_out}[3]
  \end{align*}
  \]
Example of a Slave Application as an S-Function

Suppose you run a DS2201 board and want to use its slave processor to run a 4-channel square wave generation. The S-function source file is \texttt{sfcn_ds2201\_4df.c} for the signal generation.

The corresponding Simulink model could look like this:

The DS2201-specific S-function implementing the 4-channel square wave generation could read:

```c
/***************************************************************
FILE: sfcn_ds2201\_4df.c
DESCRIPTION:
S-function to perform a 4-channel frequency generation based on the 4df slave application (DS2201 version).
Copyright (c) 1997-2005 dSPACE GmbH, GERMANY
***************************************************************************/
#define S_FUNCTION_LEVEL 2
#define S_FUNCTION_NAME sfcn_ds2201\_4df
#include "simstruc.h"
/* Include basic real-time environment header. */
/* Include the slave application source code to be found in %DSPACE_ROOT%\slave\ds2201\4df.c */
#ifndef MATLAB_MEX_FILE
#include <brtenv.h>
#include <%DSPACE_ROOT%\slave\ds2201\4df.c>
#endif
#define NUM_INPUT_PORT (1)
#define INPUT_PORT_WIDTH_1 (4)
/* Function: mdlInitializeSizes =========================== */
static void mdlInitializeSizes(SimStruct *S)
{
  /* This S-function has no parameters. */
  ssSetNumSFcnParams(S, 0); /* No. of expected parameters */
  if (ssGetNumSFcnParams(S) != ssGetSFcnCount(S)) return;
  /* Set the number of states to zero. */
  ssSetNumContStates(S, 0);
  ssSetNumDiscStates(S, 0);
  /* We have one input port with width 4*/
  static void mdlInitializeSizes(SimStruct *S) { /*
    /* We have one input port with width 4*/
```
if (!ssSetNumInputPorts(S, NUM_INPUT_PORT)) return;
ssSetInputPortWidth(S, 0, INPUT_PORT_WIDTH_1);
/* We have no output ports. */
if (!ssSetNumOutputPorts(S, 0)) return;
/* This S-function has one sample time. */
ssSetNumSampleTimes(S, 1);
/* No work vectors and other special features are used. */
ssSetNumWork(S, 0);
ssSetNumWork(S, 0);
ssSetNumWork(S, 0);
ssSetNumWork(S, 0);
ssSetNumSampledECs(S, 0);
/* No additional options are used. */
ssSetOptions(S, 0);
}
/* Function: mdlInitializeSampleTimes ===================== */
static void mdlInitializeSampleTimes(SimStruct *S)
{
/* Use 5 ms sample time and no offset. */
ssSetSampleTime(S, 0, 0.005);
ssSetOffsetTime(S, 0, 0.0);
}
#define MDL_START /* Change to #undef to remove function */
#if defined(MDL_START)
/* Function: mdlStart =================================== */
static void mdlStart(SimStruct *S)
{
#ifndef MATLAB_MEX_FILE
/* Initialize the DS2201 I/O board. */
ds2201_init(DS2201_1_BASE);
/* Load the slave application. */
load_slave_4df(DS2201_1_BASE);
#endif
#endif /* MDL_START */
/* Function: mdlOutputs ================================== */
static void mdlOutputs(SimStruct *S, int_T tid)
{
/* Get pointer to S-function input port. */
#ifndef MATLAB_MEX_FILE
int j;
InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
for (j = 0; j < INPUT_PORT_WIDTH_1; j++)
{
/* Write frequency value for channel j+1. */
ds2201_dtof_4(DS2201_1_BASE, j+1, *uPtrs[j]);
}
#else
/* do nothing */
#endif
/* Function: mdlTerminate ================================== */
static void mdlTerminate(SimStruct *S)
{
#endif /* MATLAB_MEX_FILE */
#include "simulink.c" /* MEX file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration fcn */
#endif
}
Managing the Simulation

Managing the simulation covers simulation control, data acquisition, and error handling.

Simulation Control
There are different options for controlling the simulation state. Refer to Simulation Control (RUN/STOP Mechanism) on page 195.

Acquiring data
The dSPACE boards support time-stamping, where a time stamp is assigned to each execution step of each task. Time-Stamping and Data Acquisition on page 200.

If the default data acquisition does not fulfill your requirements, you can set up a more flexible data acquisition. Refer to How to Modify Data Acquisition Behavior on page 205.

Run-time errors
RTI responds to run-time errors by stopping the simulation. Refer to Reaction to Run-Time Errors on page 208.
The results of Simulink and real-time simulations differ under certain conditions. Refer to *Different Results from Different Simulation Modes* on page 209.
Simulation Control
(RUN/STOP Mechanism)

For both RTI and RTI-MP the simulation state can be specified in a number of ways:

- It can be preset via the Initial simulation state option. When the application is compiled it inherits this setting. For details, refer to Real-Time Workshop Page (for RTI) and Main Page (for RTI-MP) in the RTI and RTI-MP Implementation Reference.

  If the initial simulation state is “PAUSE”, the output devices are set to the Initialization values. If the initial simulation state is “STOP”, they are set to the Termination values.

- It can be changed during run time via the simState variable (for details, refer to simState in the RTI and RTI-MP Implementation Reference):
  
  - Manually via ControlDesk, MLIB/MTRACE and CLIB (available in the Main group of the variable description file).
  
  - Automatically by the simulation itself via the simState SET block (see simState SET Block in the RTI reference that corresponds to your hardware).

- If an error occurs during the real-time simulation, the simulation is stopped automatically. After that the real-time application is terminated via the exit() function. See also Reaction to Run-Time Errors on page 208.

The simState variable can also be used in the simulation via the simState READ block (see simState READ Block in the RTI reference that corresponds to your hardware).
When the simulation state is being changed, the following actions are carried out:

<table>
<thead>
<tr>
<th>Change to</th>
<th>Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP</td>
<td>0</td>
<td>The termination code is executed and the <code>currentTime</code> variable reset.</td>
</tr>
<tr>
<td>PAUSE</td>
<td>1</td>
<td>The simulation is paused (the <code>currentTime</code> variable is not reset).</td>
</tr>
<tr>
<td>RUN</td>
<td>2</td>
<td>If the simulation was previously in the PAUSE mode, the execution of the application is reenabled. If the previous state was STOP, the states of the model are reset to their initial values and the execution of the application is restarted.</td>
</tr>
</tbody>
</table>

The simulation state is processed in the background task. As long as you do not reset the hardware, the background task continues running.

If you carry out ControlDesk's Stop Real-Time Processor or Simulink Simulation command, the processor is stopped immediately, which is not the same as setting the `simState` to STOP. Therefore, the termination code is not executed and the background task is no longer active. As a result, the I/O does not output the termination values of the application and you have to reload the application again via ControlDesk's Reload Application or Start Simulink Simulation command if you want to restart it. (Both commands are described in the ControlDesk Reference.)
How to Set and Read the Simulation State

You can change the simulation state during run time via the simState variable (see simState in the RTI and RTI-MP Implementation Reference):

- Manually via ControlDesk, MLIB/MTRACE and CLIB (available in the Main group of the variable description file)
- Automatically by the simulation itself via the simState SET block

To change the simulation state from ControlDesk

- Connect the simState variable from the Main group of the variable description file to a suitable instrument in your layout. For example, you can use the RadioButton instrument (see How to Prepare an Experiment for the Operator Mode in the ControlDesk Experiment Guide).

To change the simulation state from within the Simulink model

1. Add the simState SET block from RTI’s Extras library to your model (see simState SET Block in the RTI reference of your controller/processor board). Several instances of this block are allowed.
2. Open the Block Parameters dialog, and select the desired action in the Set simState to drop-down list (RUN, PAUSE, STOP). You can also choose the trigger type and whether the block should have an enable port.
Managing the Simulation

3. Connect the port(s) of the block to the signal(s) that should change the simulation state.

- If you want to change the simulation state while the simulation is paused or stopped, you have to place the involved blocks in the background task of the model.
- The simState SET block writes just the new value to the simState variable. The actual simulation state is switched in the background task. Therefore, the current computation step is always finished before the simulation pauses or stops. If the background task is complex to compute or if there is very little time for the background task, several further computation steps might be performed before switching takes place.
- With RTI-MP, the simState SET Block can only be used for the Master CPU.
- Setting the simulation state to STOP does not stop the real-time processor, but sets the application to a different condition: The termination values are written to the I/O, after which only the background task of the application remains active. In contrast, the Stop RTP command provided by ControlDesk stops the real-time processor immediately: The termination values are not written to the I/O, and the background task does not remain active.

To read the simulation state from within the Simulink model

1. Add the simState READ block from RTI's Extras library to your model (see simState READ Block in the RTI reference of your controller/processor board). Several instances of this block are allowed.
2. Connect the output port of the block. The data type is int32. The meanings of the output values are as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Simulation State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STOP</td>
</tr>
<tr>
<td>1</td>
<td>PAUSE</td>
</tr>
<tr>
<td>2</td>
<td>RUN</td>
</tr>
</tbody>
</table>
If you want to read the simulation state while the simulation is paused or stopped, you have to place the involved blocks in the background task of the model (in a foreground task the simState READ block would constantly read the simulation state RUN).
Time-Stamping and Data Acquisition

The dSPACE boards support time-stamping, where a time stamp is assigned to each execution step of each task. As a result, ControlDesk can use the actual time values to plot the signals of tasks driven by interrupt blocks. One benefit is that you can see when the aperiodic interrupts occur in relation to each other.

Generating time stamps

The time stamps are generated in different ways for each task type:

**Timer tasks**   For timer tasks, the dSPACE Real-Time Kernel calculates the time stamps from the sample time of the tasks in the same way as the Real-Time Workshop timing engine calculates the time for timer tasks (currentTime $T_t$):

$$T_t = K \cdot T_{\text{sample\_time}} + T_{\text{offset}}$$

where

- $T_t$ = currentTime variable or RTK time stamp for timer task
- $K$ = execution counter

This ensures consistency between the time of the real-time application and the time displayed in ControlDesk's plots.
Tasks driven by hardware interrupt blocks  For tasks that are driven by hardware interrupt blocks, the dSPACE Real-Time Kernel derives the time stamps ($T_{hw}$) from a hardware clock. This ensures exact time stamps that are independent of the fixed step size of the real-time application:

![Diagram showing time stamps and intervals](image)

If the real-time application also contains a timer task (this is the case for all multiprocessor applications), that task acts as a reference task for the asynchronous tasks. The dSPACE Real-Time Kernel derives the time stamps for asynchronous tasks in relation to the time stamps of the reference task to ensure time consistency, especially if a simulation runs over a long period of time:

![Diagram showing time stamps and intervals](image)
Managing the Simulation

\[ T_{hw} = T_t + \Delta T_{hw} \]

where

- \( T_{hw} \) = time of task driven by a hardware interrupt block
- \( T_t \) = \( \text{currentTime} \) variable or RTK time stamp for timer task
- \( \Delta T_{hw} \) = time that elapsed since the last increment of \( T_t \)

**Tasks driven by Software Interrupt blocks** For tasks driven by Software Interrupt blocks, the dSPACE Real-Time Kernel uses the time stamps from the task that issues the software interrupt.

The \( \text{currentTime} \) variable of the real-time simulation is calculated by Real-Time Workshop using floating-point numbers, whereas the time stamps are calculated by the dSPACE Real-Time Kernel using integer numbers. Since the precision of floating-point numbers decreases the larger the numbers become, the \( \text{currentTime} \) variable and the associated time stamps might differ slightly.

**Time-stamping and RTI-MP distributed tracing**

Via the distributed tracing feature, ControlDesk is able to capture the data individually from the different CPUs of a multiprocessor system and display the data together in a single plot window.

Distributed tracing is available only for boards that support time-stamping and multiprocessing, namely the DS1005 and DS1006 boards.
Host services  In addition, the processors of the multiprocessor system can have different host services specified by Data Capture blocks. Distributed tracing is used only with host services that have the same service name on all the processors used. The following table shows an example:

<table>
<thead>
<tr>
<th>Host Service Name</th>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>“100 µsec”</td>
<td>T = 0.0001</td>
<td>T = 0.0001</td>
<td>T = 0.0001</td>
</tr>
<tr>
<td></td>
<td>T = 0.01</td>
<td>Not available</td>
<td>Not available</td>
</tr>
<tr>
<td></td>
<td>Interrupt-block-driven task</td>
<td>Interrupt-block-driven task</td>
<td>Interrupt-block-driven task</td>
</tr>
<tr>
<td>“10 msec”</td>
<td></td>
<td>Not available</td>
<td>Not available</td>
</tr>
<tr>
<td></td>
<td>Interrupt-block-driven task</td>
<td>Interrupt-block-driven task</td>
<td>Interrupt-block-driven task</td>
</tr>
<tr>
<td>“Crankshaft”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interrupt-block-driven task</td>
<td>Interrupt-block-driven task</td>
<td>Interrupt-block-driven task</td>
</tr>
<tr>
<td>“10 degree”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interrupt-block-driven task</td>
<td>Interrupt-block-driven task</td>
<td>Interrupt-block-driven task</td>
</tr>
</tbody>
</table>

An exception to this rule applies to host services that are called in timer tasks: If the processor does not have a host service running at an appropriate rate, the host service of the fastest timer task is used instead. For example, if a variable on processor 2 in the example above is captured via the 10 msec host service, the 100 µsec host service is used instead. As a result, you can use the following host services for distributed tracing:

<table>
<thead>
<tr>
<th>Host Service Name</th>
<th>Available on Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>“100 µsec”</td>
<td>Yes</td>
</tr>
<tr>
<td>“10 msec”</td>
<td>Yes</td>
</tr>
<tr>
<td>“Crankshaft”</td>
<td>No</td>
</tr>
<tr>
<td>“10 degree”</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The host services are created by RTI and can be edited in the related model by using Data Capture blocks. If you do not include a Data Capture block in your model, there is one host service that is executed with the basic step size on each processor.

For further information on the distributed tracing feature, see Capturing Data and Distributed Tracing in Multiprocessor Systems in the ControlDesk Experiment Guide.
Limitations

For information on the limitations with the time-stamping feature, refer to Limitations on page 275.
How to Modify Data Acquisition Behavior

Suppose you want to use ControlDesk to observe variable changes that occur in a task driven by an interrupt block. If the host service code for real-time data acquisition is placed in a timer task, some variable changes might be lost as illustrated below:

In contrast, if the host service code is placed inside the task driven by an interrupt block, all variable changes from this task can be captured as illustrated below:

This effect applies especially to the task information variables:
Some variables show different behavior depending on where the host service is executed. Consider the overrunQueueCount and state variables of a particular task, for example:

- **overrunQueueCount** is always 0 if read in the background task. This is due to the fact that the background task has the lowest priority and is therefore executed only if no other task is queued. If overrunQueueCount is read from within its own task, its value is always 1 or higher because it is incremented when the task is called and decremented when the task is finished.

- **state** is always idle (0) if read in the background task. If state is read from within its own task, it is always running (2), and if read from any other task, it can be running (2), ready (1) or idle (0).

If you use one of ControlDesk's virtual instruments to access a variable, ControlDesk automatically uses the host service that is executed in the background task.

If you trace a variable via ControlDesk's data acquisition instruments, you can specify the desired host service via the Capture Settings.

The default host service code for data acquisition is executed with the fastest timer task. This code is omitted automatically if you place at least one Data Capture block in your model.

**To modify the data acquisition behavior**

1. Place up to four Data Capture blocks from RTI's Extras library in your model, and open the Block Parameters dialog.
   - To achieve a host service that is executed in a task driven by an interrupt block, place the Data Capture block inside a function-call subsystem. Specify -1 as the Sample time.
   - To achieve a host service that can be turned on or off via a control variable, place the Data Capture block inside an enabled subsystem which is not located inside a triggered or function-call subsystem. Specify the desired Sample time in the Block Parameters dialog.
• To achieve a downsampling compared to the default host service code, place the Data Capture block in your model so that it is not inside a conditionally executed subsystem. Specify the desired **Sample time**. As an alternative, you can achieve a downsampling via ControlDesk, so that you do not need to modify your model (see *How to Capture Data* in the *ControlDesk Experiment Guide*).

• To achieve a host service that is executed on specific signal conditions, place the Data Capture block inside a triggered subsystem (trigger type: "rising", "falling" or "either"). Specify -1 as the **Sample time**.

2 In the Block Parameters dialog, specify a unique **Service number** and **Service name**.

3 Select the desired service for each data acquisition instrument in ControlDesk (see *Capturing Data* in the *ControlDesk Experiment Guide*).

4 If you want to use the time-stamping feature, you have to activate it in ControlDesk (see *Capturing Data* in the *ControlDesk Experiment Guide*).

Do not place the Data Capture block in the background task or in a function-call subsystem driven by a Software Interrupt block from within the background task. This leads to unpredictable results because there are no useful time stamps for the background task.

For further information on the Data Capture block, refer to *Data Capture Block* in the RTI reference that corresponds to your hardware.
Reaction to Run-Time Errors

When you run a real-time simulation, a run-time error might occur, meaning a situation where the real-time application can no longer be executed. Such run-time errors can result, for example, from incorrect code, task overrun or user-specific function calls to certain Message Module functions or the `ssSetErrorStatus` function.

RTI responds to run-time errors by stopping the simulation. The termination code is executed and the termination values are written to the I/O.

- If the run-time error was caused by a nonfatal error such as an overrun situation, the application remains in the simulation state STOP and can be restarted (see How to Set and Read the Simulation State on page 197).

- All other run-time errors are most likely fatal errors. Therefore, RTI also terminates the real-time application via the `exit()` function. As a result, you cannot restart the application via the simulation state, but you have to reload the application.
Different Results from Different Simulation Modes

With RTI and RTI-MP, you can simulate your model in different modes. The following simulation modes are available:

- Simulink simulation
- Real-time simulation

For details on the simulation modes, refer to Control design and Simulink simulation on page 18 and RCP with dSPACE Prototyper on page 18.

Reasons for different simulation results

The results of the Simulink simulation provide a reference for real-time simulation results. However, there can be differences between the results from different simulation modes. This chapter informs you of the problems that can arise when you run simulations in different modes. Due to the complexity of this issue, it is not possible to give precise instructions on preventing or minimizing the differences. You should be aware of the facts given below when comparing the results of Simulink and real-time simulation:

- Different behavior caused by interrupt-driven tasks
- Different rules for modeling
- Different task scheduling in multiple timer tasks mode
- Different output values of I/O blocks
- Incongruent signals from signal generators

Different behavior of interrupt-driven tasks

In Simulink simulation mode, modeled interrupt-driven tasks are executed in a different way than in real-time simulation. The Simulink simulation executes these tasks in fixed steps of periodic tasks. That means they are executed when the RTI interrupt block connected to the function-call port of a subsystem is executed by Simulink. There are also no task interrupts in Simulink simulations.
Different rules for modeling
Suppose you use Simulink Function Call Generator blocks instead of RTI Interrupt blocks in your model when simulating it in Simulink Simulation mode. You are not allowed to use RTI Task Transition blocks because Simulink does not identify interrupt-driven tasks. However, when you later replace the Simulink Function Call Generator blocks with RTI Interrupt blocks for real-time simulation, Simulink identifies interrupt-driven tasks and forces you to implement valid task transition using the RTI Task Transition blocks. You have to change your model, so its simulation behavior is also different.

Multiple timer tasks mode
In Simulink simulation, the tasks are executed one after another. In real-time simulation, tasks can be preempted by higher priority tasks. Thus, in real-time simulation mode, a task can interrupt another task it exchanges signals with. In this case, signals can differ from signals in Simulink simulations.

Output of I/O blocks
You have to use RTI I/O input blocks to connect the real-time simulation to the physical world. In real-time simulations, these blocks provide valid output values. If you perform a Simulink simulation, however, the RTI I/O blocks do not read physical values. Thus, their output values are always 0.

Incongruent signals of signal generators
The signals of signal generators may not be exactly congruent in Simulink simulation mode and in real-time simulation mode. For example, in real-time simulation mode, an edge of a square wave may occur one calculation step earlier or later than in Simulink simulation mode. The algorithms which are calculated in the two simulation modes are not 100% identical. Thus, different rounding errors occur. This basically affects every Simulink block. However, for a square wave generator where an edge shifts by one sampling step, the effect of the different rounding errors is more easily seen than for other blocks.
Advanced Techniques

You might want to use the following advanced techniques to achieve optimized simulation.

**Atomic subsystems**
Atomic subsystems can help you optimize the build process and simulation. Refer to *Optimizing the Build Process and Simulation via Atomic Subsystems* on page 213.

**Excluding subsystems from the TRC file**
Using the TRC Exclusion block it is possible to reduce the size of the generated variable description file (TRC file). By excluding large parts from the TRC file, you can reduce the time for building the TRC file and loading it in ControlDesk. The block also allows you to hide model parts that are not to be seen by other users. Refer to *Excluding Subsystems from the TRC File* on page 218.
Applying Subsystem permissions in the TRC file

By applying subsystem permissions to the TRC file, Simulink’s ReadOnly and NoReadOrWrite permissions for subsystems affect not only the Simulink model itself but also the experiment software, such as ControlDesk. As a result, all the block parameters of ReadOnly subsystems are read-only in ControlDesk, and the content of NoReadOrWrite subsystems is hidden in ControlDesk. Refer to How to Apply Subsystem Permissions to the TRC File on page 221.

Block priorities

Via the priority of Simulink blocks, you can achieve a user-specific execution order for the model. Refer to Modifying the Block Execution Order on page 223.

Blocks requiring additional parameter tuning steps

Certain blocks require additional steps or different handling of tuning parameters via ControlDesk, because parameters can be only partly tuned, or not tuned at all, using the default techniques.

- The parameters of Transfer Fcn, Zero-Pole, and State-Space blocks are not tunable by default. You can choose one of three methods to enable parameter tuning. Refer to Tuning Parameters of Transfer Fcn, Zero-Pole and State-Space Blocks on page 226.

- The parameters of multi-dimensional Look-Up Table (n-D), Direct Look-Up Table (n-D), and Interpolation (n-D) using PreLook-Up blocks can only be tuned via multiple 2-D slices. The handling of 1- and 2-dimensional blocks also slightly differs from that of 1-D and 2-D Look-Up Table blocks. Refer to Tuning Parameters of n-D Look-Up Table Blocks on page 232.

Migrating from AlphaCombo

The AlphaCombo system is no longer supported. You can migrate your AlphaCombo models either to a multiprocessor system or to a single-processor system based on the DS1005 or DS1006. Refer to How to Migrate from AlphaCombo to DS1005 or DS1006 on page 234.

Using Simulink’s Model Verification blocks

You might want to verify certain signals of your simulation and issue a warning or even stop the simulation if the signals leave the specified range. Refer to How to Use Simulink’s Model Verification Blocks on page 238.
Optimizing the Build Process and Simulation via Atomic Subsystems

Atomic subsystems are a feature of Simulink that can help you optimize the build process and simulation.

**Basics**
Simulink and Real-Time Workshop distinguish between virtual and atomic subsystems. Refer to Virtual Versus Atomic Subsystems on page 213.

**Bundling block execution**
You can use atomic subsystems to bundle the execution of blocks. Refer to How to Bundle the Execution of Blocks on page 214.

**Optimizing the compilation**
You can use atomic subsystems to optimize compilation. Refer to How to Reduce the Function and File Sizes via Atomic Subsystems on page 216.

**Related Topics**
See also Using Simulink by The MathWorks.

Virtual Versus Atomic Subsystems

Simulink and Real-Time Workshop distinguish between virtual and atomic subsystems.

**Virtual subsystems**
Virtual subsystems are mainly used to structure a model. In contrast to atomic subsystems, it makes no difference to the function of the model whether a block is located in a virtual subsystem or in the parenting subsystem. The code that is generated for a virtual subsystem is embedded in the code of its parenting subsystem. This is also true for nested virtual subsystems. The code generated for blocks inside a virtual subsystem can interleave with code generated for blocks outside it.
Atomic subsystems  In contrast to virtual subsystems, these can be used to influence the block execution order: The code generated for atomic subsystems does not interleave with code generated for blocks outside those subsystems. The resulting code resembles an unconditionally inlined function, which is called a code block. A pure atomic subsystem is always implemented as a code block. In the Simulink model, atomic subsystems are marked with a bold block frame.

RTI does not fully support the code reuse feature for nonvirtual subsystems provided by Real-Time Workshop. If you want to enable parameter tuning for mask and workspace parameters, you cannot use the code reuse feature. For details, refer to General Limitations on page 275.

Triggered and enabled subsystems  Simulink automatically specifies triggered and/or enabled subsystems as atomic and disables the Treat as atomic unit checkbox for them, so the specification cannot be changed.

Subsystems driven by an RTI interrupt block  Subsystems that are triggered by an RTI Interrupt block are always atomic. They are always generated into separate functions.

How to Bundle the Execution of Blocks

You might want to bundle the execution of certain blocks so that they are computed as a code block which is not interleaved with code from other subsystems. The execution of the code block still can be interrupted by higher-priority tasks.

To bundle the execution of blocks

1  Place the blocks to be bundled in a subsystem and open the Block Parameters dialog of the subsystem.

2  Select the Treat as atomic unit checkbox to make the subsystem atomic.

The atomic subsystem is marked with a bold block frame.
Code reuse feature

Due to a limitation of Real-Time Workshop, you cannot use the reusable function option for nonvirtual subsystems if you want to have mask and workspace parameters generated into the variable description file. In addition, reused subsystems are not included in the variable description file. Therefore, the dSPACE installation disables the code reuse feature for dSPACE RTI targets by default. This means that non-virtual subsystems are not generated as reusable functions regardless of the settings you specify for the subsystems of your models.

If you want to enable the code reuse feature (not recommended), carry out the following instructions:

To enable the code reuse feature

1. Make sure that mask and workspace parameters are not generated into the variable description file:

   - For RTI models, open Real-Time Workshop page of the Simulation Parameters dialog, change to the “RTI variable description file options” category and clear the Include mask and workspace parameters checkbox.
   - For RTI-MP models, open a CPU's page of the Multiprocessor Setup dialog, then open the Variable Description File Options page of the CPU Options dialog and clear the Include mask and workspace parameters checkbox. This is a global option that affects all CPUs.

2. To enable the code reuse feature, change to the MATLAB Command Window and enter

   `rti_option EnableRTWCodeReuse 1`

   The code reuse feature for non-virtual subsystems is then enabled for all models until you carry out the command:

   `rti_option EnableRTWCodeReuse 0`
How to Reduce the Function and File Sizes via Atomic Subsystems

Compilers are usually limited with regard to the code size (number of lines) of functions and source code files. A large source code file may also have a significantly longer compilation time than several small files. Therefore, you should split large functions into smaller ones and also distribute the contents of a large source code file to several smaller files.

In addition to the characteristics pointed out above, atomic subsystems and triggered and enabled subsystems allow you to achieve smaller functions and source code files.

To reduce the function size via atomic subsystems

1. Select the standard subsystem you want to optimize and open its Block Parameters dialog.
2. Select the Treat as atomic unit checkbox.
   The RTW system code drop-down list is enabled.

   Triggered and enabled subsystems are always atomic and the Treat as atomic unit checkbox is disabled.

3. Select “Function” as the RTW system code.
   As a result, Real-Time Workshop does not inline the code for the subsystem into the MdlOutputs() function of the parenting subsystem but creates a separate function. Therefore, the function size is reduced. However, the file size remains nearly constant.

   It is not possible to change the generation of functions for subsystems driven by RTI interrupt blocks: The settings relevant to the RTW System code option are ignored.

   The RTW function name and RTW file name options are enabled.
To reduce the source code file size via atomic subsystems

1. Carry out the instructions To reduce the function size via atomic subsystems on page 216.

2. Select something other than “Auto” as the RTW file name option.

As a result, Real-Time Workshop writes the function to a separate source code file and thus reduces the size of the `<model>.c` file. Depending on your selection, the RTW file name edit field might be enabled. If this is the case, enter the desired RTW file name without an extension.
Excluding Subsystems from the TRC File

Excluding a subsystem from the TRC file is an optimization technique provided by RTI.

Basics
There are two reasons why you might want to exclude a subsystem from the generation of the TRC file. Firstly, to reduce the size of the TRC file, or secondly, to hide parts of a model from other users. Refer to Advantages of a Reduced TRC File on page 218.

Excluding a subsystem
When a TRC Exclusion block is added to a subsystem, all blocks in the subsystem where the block resides and all underlying subsystems are not generated into the TRC file. Refer to How to Exclude a Subsystem from the TRC File on page 220.

Advantages of a Reduced TRC File

There are two reasons why you might want to exclude a subsystem from the generation of the TRC file. Firstly, to reduce the size of the TRC file, or secondly, to hide parts of a model from other users.

Reduced file size
Normally, a generated TRC file contains all the signals defined in the model, which in the case of large models can result in a large TRC file. Subsequently, such a file requires more time for generation, which in turn results in increased loading times in ControlDesk. By adding a TRC Exclusion block to your model at a selected location, you can exclude all the signals from the blocks in the selected subsystem as well as all underlying subsystems. This means the signals are not included in the TRC file when it is generated, thus reducing the size of the file.
Hiding model parts

You can also use the TRC Exclusion block to hide parts of a model from other users. Since the block signals of the subsystem and all underlying subsystems are excluded from the TRC file when the TRC Exclusion block is active, they are not available for selection in the Variable Browser in ControlDesk.

The illustration above shows a subsystem with a TRC Exclusion block set inactive, which is the same as if the block was not present at all. As you can see, the Variable Browser Basics in ControlDesk displays all the block groups of the subsystem. Now take a look at what happens when you set the block active.

The TRC Exclusion block is now set active. The Variable Browser no longer displays the block groups of the subsystem. The only group available is the subsystem itself. All underlying subsystems are also not visible.

You can add more than one TRC Exclusion block to a model, which gives you full control over the parts of the model that are to be hidden. An extreme scenario would be to add the TRC Exclusion block to the root of the model. This would result in an empty Model Root group in the TRC file.
The mode of the TRC Exclusion block is set via the block dialog. You can enter a value directly or via a workspace variable. By using workspace variables, you could define different generation patterns or view scenarios and switch between them according to your needs.

How to Exclude a Subsystem from the TRC File

When a TRC Exclusion block is added to a subsystem, all blocks in the subsystem where the block resides and all underlying subsystems are not generated into the TRC file.

To exclude a subsystem

1. Open the subsystem where the TRC Exclusion block is to reside.
2. Add the TRC Exclusion block from the Extras library to the subsystem.

The block mode in the block dialog is set to 1 (active) by default. You can switch the block inactive by opening the block dialog and setting the block mode to 0.
How to Apply Subsystem Permissions to the TRC File

By applying subsystem permissions to the TRC file, Simulink's ReadOnly and NoReadOrWrite permissions for subsystems affect not only the Simulink model itself but also the experiment software, such as ControlDesk. As a result, all the block parameters of ReadOnly subsystems are read-only in ControlDesk, and the content of NoReadOrWrite subsystems is hidden in ControlDesk.

Basics

In RTI and RTI-MP you can specify via the **Apply subsystem read/write permissions** option whether the subsystem permissions are reflected in the experiment software.

Restrictions

- If you also select the **Include mask and workspace parameters** option, changing mask or workspace parameters during real-time simulation affects all dependent parameters regardless of whether their parent subsystem is ReadOnly or NoReadOrWrite. This behavior is required to preserve the integrity of the model parameterization.
- If ControlDesk is configured to allow browsing into masked and NoReadOrWrite subsystems, it does not hide the NoReadOrWrite subsystems. However, the parameters of such subsystems remain read-only.
- Although the **Apply subsystem read/write permissions** option originates from RTI and RTI-MP dialogs, it also affects the TRC files for ControlDesk-controlled Simulink simulations.

Possible methods

- If your model is an RTI model, see **To apply subsystem permissions to the TRC file for RTI models** on page 222
- If your model is an RTI-MP model, see **To apply subsystem permissions to the TRC file for RTI-MP models** on page 222
To apply subsystem permissions to the TRC file for RTI models

1. Open the Real-Time Workshop page of the Simulation Parameters dialog and select the “RTI variable description file options” category.

2. Select the Apply subsystem read/write permissions option.

To apply subsystem permissions to the TRC file for RTI-MP models

1. Open a CPU’s page of the Multiprocessor Setup dialog and click the Variable Description File Options button.
   The CPU Options dialog opens on the Variable Description File Options page.

2. Select the Apply subsystem read/write permissions option.

Now the TRC file is generated using the read/write permissions of subsystems.

Next steps

If desired you can now configure ControlDesk to show NoReadNoWrite subsystems. Refer to Using Workspace and Mask Parameters in the ControlDesk Experiment Guide.
Modifying the Block Execution Order

By specifying block priorities, you can modify the execution order of the blocks in the overall execution of the Simulink model.

**Basics**
The priority of Simulink blocks can be specified in different ways. Refer to *Determining the Execution Order* on page 223.

**Checking for conflicts**
Illogical distribution of block priorities can cause conflicts. Refer to *How to Check for Illogical Block Priorities* on page 225.

**Determining the Execution Order**

The priority of Simulink blocks and thus their execution order can be specified by means of:

■ The `SS_OPTION_PLACE_ASAP` option in a level 2 S-function
■ The block priority

**SS_OPTION_PLACE_ASAP**

Level 2 S-functions are computed as soon as possible by setting the `ssSetOption(SS_OPTION_PLACE_ASAP);` option in the `mdlInitializeSizes` function.

**Block priority**

You can specify a priority for a block in its Block Parameters dialog. This allows you to influence the block execution order as long as there is no topological dependency. You can specify any integer value – even negative numbers and zero: the lower the number, the higher the priority of the block.

If you run an RTI-MP model, you might cause priority conflicts by assigning block priorities. Such conflicts occur between user-specific and RTI-MP-specific blocks in the extracted submodels. See *How to Check for Illogical Block Priorities* on page 225 for information on how you can avoid them.
Resulting execution order
First, Simulink tries to satisfy the level 2 S-function demand to place the block as soon as possible (meaning that the SS_OPTION_PLACE_ASP option has the highest priority). If a model contains multiple level 2 S-function blocks with the SS_OPTION_PLACE_ASP option, the associated block priorities are used to determine which of these blocks is executed first. After that Simulink looks for user-specific block priorities in the model.

Priority conflicts
Due to topological dependencies or internal conflicts, MATLAB might not be able to apply user-specific block priorities. In that case, a warning is issued, and the conflicting priorities are ignored.

Priority conflicts with RTI-MP
When extracting the submodels of the individual CPUs, RTI-MP automatically sets block priorities of 10,000 for all IPC Input blocks and -10,000 for all IPC Output blocks to achieve the smallest possible waiting times for the interprocessor communication.

To satisfy the data dependencies, Simulink ignores the block priorities for direct feedthrough terms. As a result, the IPC Input block is calculated before the IPC Output block, and MATLAB issues a warning.

You should be careful when assigning user-specific priorities to blocks because the priorities might conflict with the priorities of the IPC Input/Output block in the extracted submodels.

User-specific block priorities might influence the sequence of interprocessor communication, which might lead to a deadlock, even if they do not cause a warning! So be careful when you work with the SS_OPTION_PLACE_ASP option in S-functions or with block priorities in your multiprocessor model.

Since such priority conflicts appear only in the extracted submodels, you should always check the submodels of all the CPUs for priority conflicts if you specify user-specific block priorities.

For information on how RTI-MP optimizes model evaluation via block priorities, see Model Evaluation on page 133.
How to Check for Illogical Block Priorities

Whenever you specify user-specific block priorities in a model, you should check it for possible priority conflicts. For single-processor systems, you have to check the model itself for priority conflicts, whereas for multiprocessor systems, you have to check the submodels of the individual CPUs.

To check a model or submodel for possible priority conflicts

1. If you run a single-processor system, you can continue with step 2. Otherwise, extract the submodel you are interested in via the Show submodel button on the corresponding CPU’s page.
2. Start the Simulink simulation of the model or extracted submodel to check if Simulink issues any warnings, or simply select the Edit – Update Diagram command from the menu bar.
Tuning Parameters of Transfer Fcn, Zero-Pole and State-Space Blocks

Linear, time-invariant systems are mostly described as either transfer functions or state-space representations. MATLAB and Simulink support both mathematical forms.

When Real-Time Workshop generates real-time code, all transfer functions are automatically transformed into state-space representations. As a result, the transfer function parameters (polynomial coefficients or poles and zeros) defined in Simulink are not available in the real-time simulation. Instead the real-time code contains the matrix values of the state-space vectors and matrices (A, B, C and D). It is not very practical trying to correlate both sets of parameters, especially because the transformations carried out by Real-Time Workshop are not described within the documentation of The MathWorks.

Moreover, Real-Time Workshop optimizes the code generated for state-space representations by eliminating any zeros of the system matrices. As a result, you cannot tune parameters that have been zero when building the model.

Basics
You might want to read up on the theory of transforming a transfer function to its state-space representation. Refer to MATLAB’s State-Space Representation for a Transfer Function on page 227.

Tuning parameters
You can use the following techniques to enable the parameter tuning for Transfer Fcn and Zero-Pole blocks:

- You can manually derive the new values of the state-space matrices from the transfer function. Refer to How to Tune Transfer Function Parameters via Manual Conversion on page 229.
- You can simulate the model in the external mode. Refer to Tuning Transfer Function Parameters via External Simulation on page 229.
- You can use parameter inlining for tuning transfer function parameters like any other parameter. Refer to How to Tune Transfer Function Parameters via Parameter Inlining on page 230.
Avoiding optimization

Real-Time Workshop tries to optimize out unnecessary parameters of State-Space blocks. However, for parameter tuning you might want to keep all parameters. Refer to *How to Keep Zero Parameters of State-Space Blocks* on page 231.

**MATLAB’s State-Space Representation for a Transfer Function**

Depending on the chosen state-space variables, each transfer function can have several or even an infinite number of corresponding state-space representations. To achieve a definite result, a particular form of state-space representation is used, for example, the normalized form.

The conversion from transfer function to state-space representation can be performed via the `tf2ss` MATLAB function. This returns the state-space matrices in the *controller canonical form*.

Consider the following general transfer function in polynomial form (system of (n-1)th order) with the coefficient numbering following MATLAB’s conventions:

\[
F(s) = \frac{y(s)}{u(s)} = \frac{a_1 s^{m-1} + a_2 s^{m-2} + \ldots + a_m}{b_1 s^{n-1} + b_2 s^{n-2} + \ldots + b_n}
\]  

(1)

If you use the `tf2ss` function to convert this transfer function into a state-space representation, the following results:

\[
\dot{x} = A \cdot \dot{x} + B \cdot u
\]

\[
\chi = \begin{bmatrix} b_2 & b_3 & \ldots & b_n \\ b_1 & b_1 & \ldots & b_1 \\ 1 & 0 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \chi + \begin{bmatrix} 1 \\ 0 \\ \ldots \\ 0 \end{bmatrix} \cdot u
\]

(2)

\[
y = C \cdot x + d \cdot u
\]

\[
y = \begin{bmatrix} 0 & \ldots & 0 & \frac{a_1}{b_1} \ldots \frac{a_m}{b_1} \\ \frac{a_1}{b_1} & \ldots & \frac{a_m}{b_1} \\ \frac{a_1}{b_1} & \ldots & \frac{a_m}{b_1} \end{bmatrix} \cdot \chi + \frac{a_1}{b_1} \cdot u
\]

(3)
d is only different from zero if the degree of the numerator polynomial is equal to the degree of the denominator polynomial. In all other cases, d equals zero.

Consider the following model:

Applying the conversion derived above will lead to the following state-space representation of this system:

\[
A = \begin{bmatrix}
-4/3 & -5/3 & -6/3 & -7/3 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{bmatrix}, \quad 
C = \begin{bmatrix}
0 & 0 & 1/3 & 2/3
\end{bmatrix}
\]

B always has the form given above and d equals 0 in this case.

The code-generating process of Real-Time Workshop introduces the two vectors A and C (see TRC file), which can be accessed via ControlDesk. In ControlDesk these parameters are referenced by

\[
A[1, 1] = -4/3 \quad \quad \quad \quad \quad C[1, 1] = 0 \\
A[2, 1] = -5/3 \quad \quad \quad \quad \quad C[2, 1] = 0 \\
A[3, 1] = -6/3 \quad \quad \quad \quad \quad C[3, 1] = 1/3 \\
A[4, 1] = -7/3 \quad \quad \quad \quad \quad C[4, 1] = 2/3
\]

All the elements that are zero or one have been optimized out by Real-Time Workshop.
How to Tune Transfer Function Parameters via Manual Conversion

The most obvious but least convenient method is to derive the new values of the state-space matrices from the transfer function by applying the conversion rules shown in MATLAB’s State-Space Representation for a Transfer Function on page 227 or by using MATLAB’s built-in tf2ss function, which directly calculates all matrix positions.

To convert a transfer function to a state-space description

1 In the MATLAB Command Window, enter the following command:
   \[ [A, B, C, D] = \text{tf2ss}(\text{NUM, DEN}) \]
   using your desired numerator and denominator.
   Enter \text{help tf2ss} to get further information on the command and its syntax.

2 Add the State-Space block from the Simulink library to your model and open its Block Parameters dialog. Set the parameters according to the results calculated via the \text{tf2ss} function.

The state-space matrices of the State-Space block (A, B, C and if it exists, D) are available in the TRC file and can therefore be tuned via ControlDesk.

Tuning Transfer Function Parameters via External Simulation

A convenient way of parameter tuning in Transfer Fcn blocks is to simulate the model in the external mode. In this simulation mode, you can change parameters in Simulink and they are downloaded to the real-time simulation. Refer to How to Run an External Simulation on page 269.
How to Tune Transfer Function Parameters via Parameter Inlining

It is possible to use parameter inlining for tuning transfer function parameters like any other parameters. This technique allows you to access the transfer function parameters directly via ControlDesk.

To use parameter inlining for Transfer Fcn blocks

1. Place the vectors of the polynomial coefficients in the MATLAB workspace, for example,

   \[
   \begin{align*}
   \text{myNum} &= [1 \ 2] \\
   \text{myDenum} &= [3 \ 4 \ 5 \ 6 \ 7]
   \end{align*}
   \]

2. Specify the workspace parameters (for example, \text{myNum} and \text{myDenum}) in the Block Parameters dialog of the Transfer Fcn block.

3. Select the Inline parameters checkbox on the Advanced page of the Simulation Parameters dialog.

   Open the Model Parameters Configuration dialog to specify tunable parameters.

   With RTI-MP the tunable parameters are not CPU-specific. Tunable parameters declared in the overall model are used for all submodels.

   For details, refer to Model Parameters Configuration Dialog (for RTI) and Model Parameters Configuration Dialog (for RTI-MP) in the RTI and RTI-MP Implementation Reference.

4. In the Source list select the referenced workspace variables you want to tune, for example, “myNum” and “myDenum” and add them to the Global (tunable) parameters table.
As a result, all the numerator and denominator coefficients in the generated code are implemented as variables that can be changed during real-time simulation.

Changing the highest-order coefficient of the denominator of a transfer function to zero causes a change in the model structure, which always requires that you rebuild the real-time code. If you changed this coefficient to zero, a run-time error such as division by zero could result. Changing zero coefficients to non-zero has no effect on the real-time simulation because they are not implemented in the real-time code.

How to Keep Zero Parameters of State-Space Blocks

Real-Time Workshop tries to optimize out unnecessary parameters of State-Space blocks. However, for parameter tuning you might want to keep all parameters. There is a simple workaround for keeping the parameters of the system matrices of a State-Space block.

To generate zero parameters of a State-Space block into the real-time code

1. Enter nonzero values for all parameters of the State-Space block.
2. Set the Initial simulation state to “STOP”.
   Refer to Real-Time Workshop Page (for RTI) and Main Page (for RTI-MP) in the RTI and RTI-MP Implementation Reference.
3. Build and download the model.
4. Download the desired parameters and switch the simulation state to run, for example, using ControlDesk.
Tuning Parameters of n-D Look-Up Table Blocks

The parameters of multi-dimensional Look-Up Table (n-D), Direct Look-Up Table (n-D), and Interpolation (n-D) using PreLook-Up blocks can only be tuned via multiple 2-D slices. The handling of 1- and 2-dimensional blocks also slightly differs from that of 1-D and 2-D Look-Up Table blocks.

Real-Time Workshop transforms the table parameters of the n-D table blocks (Look-Up Table (n-D), Direct Look-Up Table (n-D), and Interpolation (n-D) using PreLook-Up blocks) to arrays whose dimensions typically have different dimensions from the corresponding block parameters. This also applies if you configured the table as a 1-D or 2-D table. As a result, up until dSPACE Release 4.0.x, ControlDesk’s Table Editor instrument could not handle the parameters of n-D table blocks.

Since dSPACE Release 4.1, RTI generates one or more LookUpTableData entries into the variable description file. For 1- and 2-dimensional tables, this corresponds to the table parameters of the standard Look-Up Table and Look-Up Table (2-D) blocks. Therefore, you can use ControlDesk’s Table Editor instrument as usual, connecting it with the LookUpTableData entry of the table. The other table data entry in the variable description file can be ignored.

Suppose you have a 3x4 Look-Up Table (n-D) block in your model. To edit the table data via ControlDesk, connect the LookUpTableData entry to a TableEditor instrument:
Tables with more than 2 dimensions

ControlDesk’s Table Editor instrument cannot handle tables with more than two dimensions. To let you access the parameters of such a table, RTI subdivides it into several 2-dimensional table slices. This is especially useful if you use the 3rd … nth dimension to switch between different 2-D tables.

Suppose you have a 3x4x2 Look-Up Table (n-D) block. In the variable description file, this comes out as 2 `LookUpTableData` entries of the dimensions 3x4, each of which you can connect to a separate `TableEditor` instrument.

The variable description file for the Look-Up Table (n-D) block also contains the different breakpoint sets `bp01Data` … `bpnData`. 
How to Migrate from AlphaCombo to DS1005 or DS1006

The AlphaCombo system is no longer supported. You can migrate your AlphaCombo models either to a multiprocessor system or to a single-processor system based on the DS1005 or DS1006.

**Basics**

If you open the Multiprocessor Setup dialog or a Communication Channel Setup dialog of an RTI-MP model for the AlphaCombo system, RTI-MP prompts you to migrate the model to support the multi-DS1005 or multi-DS1006 system. If you start the migration, the AlphaCombo-specific options are removed and the model is migrated to the platform support that is currently active.

You must specify the GigaLink topology via the Multiprocessor Topology Setup dialog of the model. You can also adapt the CPU names, if desired.

You are recommended to migrate your AlphaCombo models to a single DS1005 or DS1006 board:

- If your model contains a lot of I/O, consider migrating to the DS1005 platform.
- If your model contains a lot of arithmetics, consider migrating to the DS1006 platform.

Migrating your application from an AlphaCombo system to a single-processor system changes the path names for the simulation variables in the variable description file (SDF, TRC). Thus, all data connections in your experiment are lost.

For Details on the Experiment Connection Converter, refer to *Converting Experiments* in the *ControlDesk Experiment Guide*. 
Restrictions

When you migrate an application from AlphaCombo to single-DS1005 or single-DS1006, you might experience the following limitation:

**Migration without RTI-MP license** If you migrate your model simultaneously from AlphaCombo to DS1005 or DS1006 and from one dSPACE Release to another (e.g., from dSPACE Release 3.5 to dSPACE Release 4.2) and you do not have an RTI-MP license for the new dSPACE Release, you will not be able to open the RTI-MP-specific dialogs of the model after it is migrated to the new dSPACE Release. You should therefore make a note of the various options contained in the Multiprocessor Setup dialog beforehand.

Then you can migrate the model to the new dSPACE Release, and after that from AlphaCombo to DS1005 or DS1006.

Instructions

To migrate a model from AlphaCombo to DS1005 or DS1006

1. Change to the MATLAB Command Window and enter `rti1005` or `rti1006` to activate the corresponding platform.

   The RTI library opens.

2. Change to the working folder of the multiprocessor model you want to migrate and open the model.

3. Change to the MATLAB Command Window and enter `set_rti` to set the new platform as the code generation target for the model.

   RTI closes and reopens the model. The settings on the Real-Time Workshop page of the Simulation Parameters dialog (“Target configuration” category) are now valid for use with the new platform.

4. Configure the options in the Simulation Parameters dialog according to the settings provided in the Multiprocessor Setup dialog and its subdialogs. The following table lists the options that need to be changed to new values or cannot be used directly:

<table>
<thead>
<tr>
<th>Specify the RTI Setting...</th>
<th>New Value...</th>
<th>According to RTI-MP Setting...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solver page</strong></td>
<td></td>
<td><strong>Main page</strong></td>
</tr>
<tr>
<td>Solver</td>
<td></td>
<td>Solver¹)</td>
</tr>
<tr>
<td>Fixed step size</td>
<td></td>
<td>Basic step size</td>
</tr>
<tr>
<td>Mode</td>
<td></td>
<td>Scheduler mode²)</td>
</tr>
</tbody>
</table>

¹) You cannot use the `Solver` tab directly but only the `Solver` on the `Fixed step size` tab.

²) You cannot use the `Scheduler mode` directly but only the `Scheduler mode` on the `Basic step size` tab.
Close the Multiprocessor Setup dialog and delete the Multiprocessor Setup block.

Replace all the IPC blocks with direct connection lines.

Replace all the IPI blocks with direct connection lines.

Delete all Default CPU blocks.

Build your real-time application. For details, refer to Building and Downloading the Model on page 241.

When these steps have been carried out, the model is a valid DS1005 or DS1006 single-processor model.
Next steps  If you have a ControlDesk experiment for the real-time application of the old multiprocessor model, you can now migrate it to match the real-time application of the new single-processor model. For details, refer to Migrating from AlphaCombo to DS1005 or DS1006 in the ControlDesk Experiment Guide.
How to Use Simulink’s Model Verification Blocks

Since MATLAB R13, Simulink provides the Model Verification blocks. These blocks can observe the signals of the Simulink simulation and issue a warning or even stop the simulation if certain conditions occur. For details on the Model Verification blocks, refer to the Simulink documentation by The MathWorks.

RTI supports the Model Verification blocks for the real-time simulation. To control the behavior of the Model Verification blocks in the real-time simulation, use RTI’s Assertion mode setting. Depending on this, one of the following actions is carried out if a verified signal leaves the desired range, meaning that the assertion occurs:

- “OFF”: No reaction (default).
- “WARN”: Issue a warning message in ControlDesk’s Log Viewer.
- “STOP”: Issue an error message and stop the simulation, i.e. set the simulation state to STOP.

In contrast to the Simulink simulation the behavior of the real-time simulation does not depend on the Stop simulation when assertion fails setting, which is available for each Model Verification block.

To verify a signal of the simulation

1. Place and connect the desired Model Verification block to your model and specify the desired parameters.

2. Specify the initial assertion mode that is active after starting the real-time simulation via the Model Verification block control setting located on the Advanced page of the Simulation Parameters dialog:
   - You can globally “Enable all” or “Disable all” Model Verification blocks, overruling the local Enable assertion settings of the individual Model Verification blocks.
• You can select the “Use local settings” entry (default), which enables the Model Verification blocks whose **Enable assertion** setting is selected.

3 Specify the desired behavior of the real-time simulation in the event that an assertion occurs:

• **RTI** Open the Real-Time Workshop page of the Simulation Parameters dialog and select the **RTI simulation options category**.

• **RTI-MP** Open a CPU’s page of the Multiprocessor Setup dialog and open the CPU Options dialog on the Build Options page.

  Select the **Assertion mode** setting “OFF”, “WARN” or “STOP”.

If desired, you can change the Assertion mode setting while the simulation is running:

**To change the assertion mode from ControlDesk**

➤ Connect the `rtiAssertionMode` variable from the Main group of the variable description file to a suitable instrument in your layout. For example, you can use the RadioButton instrument. The following values are valid:

<table>
<thead>
<tr>
<th>Value</th>
<th>Reaction to an Assertion Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No reaction</td>
</tr>
<tr>
<td>1</td>
<td>Issue a warning message in ControlDesk’s Log Viewer.</td>
</tr>
<tr>
<td>2</td>
<td>Issue an error message and stop the simulation, i.e. set the simulation state to STOP.</td>
</tr>
</tbody>
</table>

With RTI-MP, the `rtiAssertionMode` variable is CPU-specific and available from the Main group of each submodel.

If an assertion occurs during the real-time simulation, ControlDesk’s Log Viewer will issue a message, pointing out the line concerned in the real-time application’s C source files. Via this information you can find the involved block in the Simulink model. For details, refer to **How to Find the Simulink Block Related to an Exception** on page 263.
Building and Downloading the Model

When an RTI or RTI-MP model is complete, real-time code can be generated and downloaded to the dSPACE system.

Consider the throttle control example (see page 20), and suppose you want to build, download and run its real-time application with different options, like solver type or execution mode. Or suppose you have made minor changes to the S-function or User-Code of the model, and you do not want to rebuild the entire model.

Basics

Consider some basic information concerning the build and download. Refer to Basics for the Build and Download on page 243.
Building and Downloading the Model

Follow these instructions to build and download a model:

- Customize the build process by means of various options. Refer to How to Specify Options for the Build Process on page 245.
- Start the build and download procedure for your model. Refer to How to Start the Build and Download on page 248.
- If you want to automate the build and download procedure or if you modified custom C code and want to quickly update the real-time application without generating code for the model again, refer to How to Automate the Build Procedure on page 250.
- If you modified a USR.TR.C file or a USR.SDF file, you might want to update the real-time application. Refer to How to Update TRC and SDF Files on page 253.
- If you have DS230<x> boards, you can download the corresponding applications together with the real-time application. Refer to How to Download DS230x Applications on page 254.

Debugging

If you experience problems during the build and download or simulation, you might want to debug your application. Refer to Debugging an Application on page 256.

Virtual dSPACE systems

You might want to create a model without having the dSPACE system available and prepare it for later use on the actual hardware. Refer to How to Create an Application for a Virtual dSPACE System on page 265.

User-specific files, source files and libraries

You might want to add user-specific files, source files and libraries to your application. Refer to Adding User-Specific Files, Source Files and Libraries on page 266.

Related Topics

Registering platforms

Before you can start the build and download procedure, you have to ensure that your dSPACE hardware is registered correctly in ControlDesk. Refer to Registering Platforms in the ControlDesk Experiment Guide.

Simulation control

The simulation state can be controlled from within ControlDesk and/or the Simulink model itself as described in Simulation Control (RUN/STOP Mechanism) on page 195.
Basics for the Build and Download

Here you will find some basic information relating to the build and download.

**Code format**


**File handling**

RTI and RTI-MP always generate a system description file (SDF), which contains all the relevant information about the various object files and TRC files. With ControlDesk you use this file to download the application to your dSPACE system or to access the signals and parameters of the simulation.

Several intermediate files are generated during or required by the build process. These are located in the `<[sub]model>_<target>\build` folder(s), where `<target>` is your active platform support. The current working folder contains user-supplied files (for example, `<[sub]model>_usr.*`) and the files that are needed for the real-time application (such as the real-time object, variable description file, map file, etc.).

**Working board**

**RTI**

By default, RTI downloads your real-time application to the first local (bus) processor board ds1<xxx> if RTI1<xxx> is running.

If ControlDesk’s working board matches the current RTI, this overrides RTI’s default.

**RTI-MP**

By default, RTI-MP downloads your real-time application to the working board specified in ControlDesk provided that this board matches the current RTI.

You can override ControlDesk’s working board using the Processor board name (RTI) or Multiprocessor system name (RTI-MP) and Network client options. For details, refer to *Real-Time Workshop Page (for RTI)* and *Advanced Page (for RTI-MP)* in the *RTI and RTI-MP Implementation Reference*. 
Overview of the build process

When you initiate the build process, a number of utilities are invoked sequentially to build the real-time program and download it to the hardware. The build process can be divided into two separate phases: the code generation phase and the compilation phase.

RTI-MP extracts the submodel for each CPU from the overall model and carries out a similar build process for each submodel.

Code generation phase  In this phase all the C code needed for the real-time application is generated. The make_rti command is started and performs the following steps sequentially:

■ Real-Time Workshop is called to generate an intermediate file, <model>.rtw, from the Simulink model. This file is used as the basis for code generation by the Target Language Compiler (see next step).

■ The Target Language Compiler (TLC) is invoked. It reads the <model>.rtw file as well as the TLC files provided by Real-Time Workshop and RTI to generate the C code for the model and the dSPACE I/O blocks used in the model.

■ A makefile called <model>.mk containing all application-specific macros is created from the RTI template makefile and placed in the current working folder. Following the makefile syntax, the application-specific makefile holds all the commands necessary to build the real-time program for the Simulink model and download it to the hardware.

Compilation phase  In this phase the C sources are translated into the final application, which can be executed on the dSPACE real-time hardware. The application-specific <model>.mk makefile is invoked by the dsmake make utility. The utility calls:

■ The compiler/linker, which compiles the C sources of the model and the dSPACE real-time simulation environment. It also links the object files and libraries into an executable file.

■ The dSPACE loader program, which downloads the executable file to the real-time processor, starts its execution and checks whether it was started correctly.

With RTI-MP the application is not download automatically.
How to Specify Options for the Build Process

Simulink, Real-Time Workshop, RTI and RTI-MP provide options for:
- Basic simulation settings
- Task configuration and data typing
- Code optimization options
- Configuration of the variable description file
- Non real-time simulation and processor board selection
- Make process customization
- Build and download options

For a complete list of options, see RTI Dialog Reference and RTI-MP Dialog Reference in the RTI and RTI-MP Implementation Reference.

**dSPACE single-processor systems** You have to specify the majority of real-time relevant settings in the Simulation Parameters dialog (see Simulation Parameters Dialog in the RTI and RTI-MP Implementation Reference). You can access this dialog via the Simulation – Parameters menu command.

**dSPACE multiprocessor systems** You have to specify the majority of real-time relevant settings in the Multiprocessor Setup dialog (see Multiprocessor Setup Dialog in the RTI and RTI-MP Implementation Reference). You can access this dialog via the Multiprocessor Setup block, which is located on the root level of your multiprocessor model.

**To specify options for the build process**

- Set the options for the build process. The default values are sufficient for most applications.

If you run a single-processor system, you can configure your model with RTI’s default settings via the set_rti command. Refer to How to Specify RTI-Specific Settings for a Model on page 37.
Consider the throttle control example (see Throttle Control Example on page 20), and suppose you want to build it for a MicroAutoBox 1401/1501 or a DS1005 multiprocessor system with solver type ode2, a fixed-step size of 0.00065 sec, and using the single timer task mode.

**MicroAutoBox 1401/1501** The following illustrations show the corresponding settings for the Simulation Parameters dialog (for other single board systems the settings are similar):

- The Start time has to be zero
- "Open end" simulation via inf
- Fixed-step solver only
- Desired step size for the real-time simulation
- Desired timer task mode

The various categories let you configure options for code generation and the make process

The Build button starts the build and download
DS1005 multiprocessor system  The following illustrations show the corresponding settings for the Multiprocessor Setup dialog:

- "Open end" simulation via inf
- Individual CPUs' step sizes are multiples of the Basic step size.
- Starts the build for a single CPU
- Starts the build for all CPUs
- Starts the download. Enabled if the build is complete for all CPUs.
- Name of the submodel assigned to the CPU
- Desired solver for the CPU
- Multiple times basic step size defines the CPU's step size
- Opens the dSPACE Task Configuration dialog for the submodel of the CPU
How to Start the Build and Download

Once you have specified the required options, you can start the build and download procedure.

To start the build and download

1. If not already done, start ControlDesk, and register your hardware via the Platform Manager. Ensure that the working board is specified correctly. Refer to Registering Platforms in the ControlDesk Experiment Guide.

2. Via MATLAB's cd command, change to the desired working folder.

3. Open or activate the model you want to build and download.

4. If you run a multiprocessor system, you have to generate a new <model>_tp.m file whenever you build a model for the first time. The file must be regenerated if you changed or added any GigaLink connection, or if you are building an application for a virtual hardware system that is not connected to your PC. The <model>_tp.m file contains the topology information of your multiprocessor system. You can generate it via the Multiprocessor Topology Setup dialog, which you can open via the MP network topology button on the Advanced page of the Multiprocessor Setup dialog. For details on the Multiprocessor Topology Setup dialog, refer to Multiprocessor Topology Setup Dialog in the RTI and RTI-MP Implementation Reference.

5. Start the build process:
   - With RTI, select the Tools – Real-Time Workshop – Build Model command from the menu bar of the model.
   - With RTI-MP, open the Multiprocessor Setup dialog, and select the Main page. To generate object files for all CPUs at once, click Build All. To generate an object file for a single CPU, click Build for the corresponding CPU.
When an up-to-date object file for a CPU exists, the Build button label reads Build OK. When all the object files are available, the Download button is enabled, which lets you download the real-time application.

Whenever you start a build and download, RTI and RTI-MP update the corresponding system description file (<model>.sdf). Therefore, you should not edit this file. If you want to customize the download procedure, you can use the user system description file (<model>_usr.sdf) instead. Refer to How to Download DS230x Applications on page 254.

If you run a multiprocessor system and experience an overrun right after the download, you might be able to avoid this via the IGNORE_SECOND_SIM_STEP option. Refer to Build Options Page in the RTI and RTI-MP Implementation Reference.

You can automate the build and download procedure via the rti_build or rtimp_build command. Refer to How to Automate the Build Procedure on page 250.

If you have just modified the custom C code of your model, you can recompile and download the real-time application via the rti_build or rtimp_build command. There is no need to regenerate the model code. Refer to How to Automate the Build Procedure on page 250.

With RTI-MP you can use the rtimpdiag diagnostic utility to get information on the real-time program. Refer to How to Run the RTI-MP Diagnostic Tool on page 256.
How to Automate the Build Procedure

You might want to automate the build process, especially when working with models that are very large and take a long time to build. Using the `rti_build` and `rtimp_build` commands, you can run build processes virtually unattended or even create build scripts that carry out several actions for you. In addition, the commands provide the following customization options:

- You can be prompted to check the task configuration or leave the task configuration as it is to avoid interaction.
- You can specify additional options for the make process.
- You can log the messages of the build process to a file.
- You can create a backup of all the run-time files directly after successfully building a model. This lets you restore the last successful build whenever the run-time files are deleted.
- You can delete all the temporary files created during the build process.

For RTI-MP models, some additional options are available:

- If an error occurs while building one submodel, the build process can be continued with the remaining submodels.
- You can build either all submodels or just a subset of submodels.

For a detailed description of the commands and their options, refer to `rti_build` and `rtimp_build` in the *RTI and RTI-MP Implementation Reference*.

To start the automatic build procedure

1. Open the RTI or RTI-MP model you want to build.
2. Change to the MATLAB Command Window and enter the `rti_build` or `rtimp_build` command with the desired build command and options using the following syntax:

   
   ```
   [errorFlag, errorMsg] = rti_build(mdlName, 'Command',<commands>[,<Parameter>,<Value> [...]])
   ```
   
   or:

   ```
   ```

   ```
   ```
Building and Downloading the Model

```matlab
[errorFlag, errorMsg] = rtimp_build mdlName, 'Command',<commands>[,<Parameter>,<Value>],...
```

where `<commands>` is one of:

- 'CodeGen' (or 'C' for short)
- 'Make' (or 'M' for short)
- 'Load' (or 'L' for short)
- 'CodeGenMake' (or 'CM' for short)
- 'MakeLoad' (or 'ML' for short)
- 'CodeGenMakeLoad' (or 'CML' for short)

See `rti_build` and `rtimp_build` in the RTI and RTI-MP Implementation Reference.

Your model is built according to the options you specified.

**To rebuild custom C code**

➤ If you modify the custom C code you can rebuild the real-time application with the 'MakeLoad' or 'CodeGenMakeLoad' command:

- Rebuild the entire model, especially if you changed the size information of any S-function (sample time, number of inputs, outputs, states, etc.) or the dimensions of data store memories used for the data transfer between the model and the User-Code:

  Recompile the MEX DLL files of the modified S-functions. Then use the 'CodeGenMakeLoad' command to rebuild the entire model.

- If it is sufficient to make and load the real-time application without generating new code from the model, use the 'MakeLoad' command.

**To create a backup of the generated real-time application**

1. Build the model as usual.

2. Create the `<location>` folder where you want to put the backup.
3 Change to the MATLAB Command Window and enter the following command:

```matlab
[errorFlag, errorMsg] = rti_build(<mdlName>, 'Command','BackUp','BackUpFolder',<location>)
or:
[errorFlag, errorMsg] = rtimp_build(<mdlName>, 'Command','BackUp','BackUpFolder',<location>)
```

A backup copy of the generated real-time application files that are necessary to load the application with ControlDesk is saved to the specified folder.

Any existing files in the `<location>` folder are overwritten without further notice.

**To remove the temporary files of the build procedure**

Change to the MATLAB Command Window, change to the working folder of the model and enter the following command:

```matlab
[errorFlag, errorMsg] = rti_build(<mdlName>, 'Command','CleanUp')
or:
[errorFlag, errorMsg] = rtimp_build(<mdlName>, 'Command','CleanUp',[<appl1>,<appl2>,...])
```

Via the `['ApplNames',<appl1>,<appl2>,...]` parameter-value pair of the `rtimp_build` command, you can remove the temporary files of specific applications.

The temporary files of the build procedure are removed.

If you replace the 'CleanUp' command with 'CleanUpAll', not only the temporary files are deleted, but also the generated real-time application(s). In this case, only the Simulink model and the user files will remain.
How to Update TRC and SDF Files

If you modified or created a USR.TRC file or USR.SDF file of a model that you already built real-time code for, you can update the TRC and SDF files as described below. This is especially useful, for example, if you started the build process before writing the files, or if you needed to modify the files.

To update a TRC file
➤ Change to the MATLAB Command Window and enter
  \texttt{rti\_usrtrcmerge <model>\_mdl}

To update a system description file
➤ Change to the MATLAB Command Window and enter
  \texttt{rti\_sdfmerge <model>\_mdl}

In both cases, \texttt{<model>\_mdl} denotes the model you want to update.
How to Download DS230x Applications

You might want to automatically download applications for DS230<x> boards whenever you download an RTI or RTI-MP model.

Download via SDF file

If you use one or more DS230<x> boards, you can download the DS230<x> applications via the <model>_usr.sdf user system description file of the model. RTI and RTI-MP incorporate the contents of the user system description file in the generated system description file (<model>.sdf) whenever you build the model.

To download DS230<x> applications via an SDF file

1. Start ControlDesk, and select Tools – SDF File Editor from the menu bar.
   - You can also open and modify an existing <model>_usr.sdf file.
2. Select the board type (DS230<x>) and add it. A dialog opens for you to specify the desired object file(s) for the different channels.
3. Repeat the 2nd step for each DS230<x> board.
4. Save and exit the SDF file. If you did not modify an existing <model>_usr.sdf file, you are prompted for a file name and folder. Choose the working folder and the name for the Simulink model according to the name of the model. You can also save a copy under a new name.

See also Creating System Description Files in the ControlDesk Experiment Guide.

Download via DDC file

If you use only a single DS230<x> board, you can also generate a C-coded project file (DDC file), which is downloaded automatically whenever you download the application for the corresponding processor board. This is the older method and has a few disadvantages compared to the download via SDF file. Therefore, the download via SDF file is recommended.
DDC files are limited to a single DS230<xx> board.

If you modify the DS230<xx> application or the DDC file, you have to rebuild the entire model. This might be more time consuming than if you download the DS230<xx> application via an SDF file.

To download DS230<xx> applications via a DDC file
1. If not already done, create an SDF file for the DS230<xx> board that specifies the required object files. You can do so via ControlDesk’s SDF File Editor.
2. Copy the desired DS230<xx> application (meaning the OBJ file and the SDF file) to the working folder of your model.
3. Create a DDC file by entering the following command at the MATLAB prompt:
   ```plaintext
   >> ! dds2c ds<xxxx> <ds230<xx>_app>.sdf <model_name>
   ```
   
4. Start a build process for the model.

As a result, the DS230<xx> applications are included in the object file of the model and downloaded to the DS230<xx> board whenever you download the object file to your processor board.

If you use standard DS230<xx> applications provided by the dSPACE installation, you will find the corresponding SDF files in the folders of the applications.
Debugging an Application

If you experience problems during the build and download or simulation, you might want to debug your application:

- **Analyzing RTI-MP applications**
  If you run a multiprocessor system, you might want to use the `rtimpdiag` diagnostic tool to analyze the application being executed. Refer to *How to Run the RTI-MP Diagnostic Tool* on page 256.

- **Handling exceptions**
  During real-time simulation you might experience exceptions that are caused by illegal arithmetic or other invalid operations or system states. For details, refer to *Handling Exceptions* on page 258.

How to Run the RTI-MP Diagnostic Tool

After you build and download a multiprocessor application, you might want some information on its status; for example, whether it initialized correctly. RTI-MP provides an easy-to-use MATLAB-based tool that can provide you with such a basic set of information about your multiprocessor application.

To work with the diagnostic tool `rtimpdiag`, your system requires MLIB/MTRACE.

**To start the RTI-MP diagnostic tool**

➤ At the MATLAB prompt, type `rtimpdiag` `[<model>]`.

If there is only a single SDF file in the working folder, you do not have to specify the SDF file name.

`rtimpdiag` provides detailed information on the running program, possible error codes and initialization failures. In addition, the number of simulation terminations and reinitializations, as well as the number of background cycles, evaluations of the base rate, and idle cycles are reported. Refer to *Troubleshooting* on page 283 if `rtimpdiag` reports any problems.
If you want `rtimpdiag` to provide profiling information for the interprocessor communication that is operated with the synchronized swinging buffer protocol, you have to specify the **Profile Synchronized Swinging Buffer Communication** option for the involved CPUs (this option is available on the CPU's page of the Multiprocessor Setup dialog). As a result, `rtimpdiag` includes information on how often data was read, the associated waiting times and whether the waiting times caused an overrun situation.

`rtimpdiag` displays a table similar to the following one in the MATLAB Command Window.

```
>> rtimpdiag
Current connection = 'BUS'.
CPU name : master  slave
Board name: ds1005 ds1005_2

Program status : started  started
Basic hardware initialization : ok  ok
Initialization of I/O Boards : ok  ok
Installation of interprocessor communication : ok  ok
Registration of Simulink model : ok  ok

Application successfully initialized : ok  ok
Error code : 0  0

Simulation terminated : 1 x  1 x
Simulation re-initialized : 1 x  1 x

Number of cycles in background process : 2742740 3067065
Number of evaluations of the base sample rate : 215792 216537
Number of idle cycles : 41907 41907
```

- The output lines from **Basic hardware initialization** up to **Application successfully initialized** indicate which initialization steps have been completed successfully by the application.
- Errors during initialization are frequently caused by I/O boards that have been used in the models but cannot be accessed on the hardware.
- `rtimpdiag` also displays how many times the simulation has been stopped and restarted.
- After a successful initialization, some diagnostic counters are always incremented by RTI-MP. These counters are displayed in the last three lines of the `rtimpdiag` table.
Building and Downloading the Model

- The first counter displays the number of cycles that have been performed in the background task. If this counter stops, the real-time processor might have stopped or the application might not respond.

- The second counter is incremented each time one step of the Simulink model is performed in the base rate service routine, which corresponds to the fastest timer task.

- If the model is in the STOP or PAUSE state, the idle counter is incremented instead of the base rate cycle counter.

If you specified the Profile synchronized swinging buffer communication option, an output similar to the following results:

```plaintext
CPU name: master

<table>
<thead>
<tr>
<th>IPC channel</th>
<th>input calls</th>
<th>wait time</th>
<th>max. wait time [usec]</th>
<th>overrun (caused by waiting)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>41192</td>
<td>0.5</td>
<td>1.85</td>
<td>no</td>
</tr>
</tbody>
</table>

CPU name: slave

<table>
<thead>
<tr>
<th>IPC channel</th>
<th>input calls</th>
<th>wait time</th>
<th>max. wait time [usec]</th>
<th>overrun (caused by waiting)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>41937</td>
<td>2.8</td>
<td>3.6</td>
<td>no</td>
</tr>
</tbody>
</table>
```

Handling Exceptions

The exception mechanism allows the processor to change to the supervisor state as a result of external signals, errors or unusual conditions arising in the execution of instructions. When exceptions occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address predefined for each exception. The exception message is reported by ControlDesk. The application stops in most cases when an exception occurs. With the help of RTLib functions, exceptions can be handled in such a way that the system is set to a predefined state before terminating.

Exception mechanism

The CPU can generate exceptions if it is not able to complete a given instruction. By default, arithmetic exceptions are disabled so that the CPU continues the computation. However, the computation results are unpredictable. It is not possible to make the processor correct arithmetic exceptions.
Localization of an error

When an exception occurs (e.g. data access error, alignment error), sometimes the error message includes the address that the last operation was executed on. The location of the address may be a clue as to which function provoked the error. However, this is not necessarily the underlying cause of the run-time error, which may have occurred in a previous function.

Information about the function associated with the address are available from the MAP file, which is generated at each compilation. However, it is not possible to find all the addresses, because the functions are indicated by their start addresses only. To find an exception that occurred within a function is very difficult, because the functions are not sorted according to location, but alphabetically (DS1005) or according to addresses (DS1006). Line numbers for identifying the instruction in the source code are not displayed.

However, there is a way to get detailed information about the location of the exception.

Debugging

To debug an application, refer to How to Trace an Exception on page 259.

Finding blocks

Real-Time Workshop writes system/block identification tags in the generated code. The tags are designed to help you identify the block in your source model that generated a given line of code. Refer to How to Find the Simulink Block Related to an Exception on page 263.

How to Trace an Exception

It is possible to identify the Simulink block that is responsible for a particular exception. First you have to identify the relevant source code in the way described below.

To find the relevant source code for an exception

1. Recompile your application using the -g option of the Microtec PPC compiler (boards with PowerPC) or DS1006 compiler. This option adds line number debug information to your generated PPC file or X86 file.

See also How to Automate the Build Procedure on page 250.
2. Download the application via ControlDesk. ControlDesk issues the program address of the exception.

3. Invoke the GNU utility `ppcobjdump` located in `%PPC_ROOT%\bin` (for boards with PowerPC processor) or `x86-objdump` located in `%X86_ROOT%\bin` (for the DS1006) to disassemble the object file and retrieve line number debug information from your application:

   - **PowerPC**: `ppcobjdump -S -l -D applname.ppc > debug.dmp`
   - **DS1006**: `x86-objdump -S -l -D applname.x86 > debug.dmp`

This command generates a `debug.dmp` file containing the following information for each source code line of the application:

- Line number
- Function name
- Module name
- Source code
- Assembler listing
- Program addresses

The source code can be printed to the dump file only if the source file and the PPC file (boards with PowerPC) or X86 file (DS1006) are in the same folder.

For detailed information about `ppcobjdump`, refer to `ppcobjdump` in the `RTLib Reference` of your PowerPC board.

For detailed information about `x86-objdump`, refer to `x86-objdump` in the `DS1006 RTLib Reference`.
4 **For DS1006:** Analyze the MAP files to find the function where the exception occurred (via its absolute start address).

Calculate the offset of the exception to the function’s absolute start address via the formula:

\[
\text{Offset}_{\text{Exception}} = \text{AbsoluteAddress}_{\text{Exception}} - \text{AbsoluteAddress}_{\text{Function}}
\]

Search the `debug.dmp` file for the function name and retrieve the relative start address of the function.

Calculate the relative address of the exception by adding its offset to the relative start address of the function.

Finally search the `debug.dmp` file for the relative address of the exception to find the corresponding source code line and the source code itself.

**For boards with PowerPC:** Search the `debug.dmp` file for the address where the exception occurred (see step 2). For this address you will find the source code line and the source code itself whenever possible.

The following lines are an excerpt from a possible output file for a board with PowerPC (in this case a DS1005). For a DS1006, the assembler commands are different and the addresses are relative.

```
0000481c <MdlOutputs>:
MdlOutputs():
smd_1005_sl.c:67
/* Compute block outputs */
void MdlOutputs(int_T tid)
{
  481c: 7c 08 02 a6 mflr r0
  4820: 94 21 ff d0 stwu r1,-48(r1)
  4824: 93 61 00 1c stw r27,28(r1)
  4828: 93 81 00 20 stw r28,32(r1)
  482c: 93 a1 00 24 stw r29,36(r1)
  4830: 93 c1 00 28 stw r30,40(r1)
  4834: 93 e1 00 2c stw r31,44(r1)
  4838: 90 01 00 34 stw r0,52(r1)
  483c: 3f 80 00 05 lis r28,5
smd_1005_sl.c:69
```
Building and Downloading the Model

```c
if (ssIsContinuousTask(rtS, tid)) {
    4840: 80 9c 99 48 lwz    r4,-26296(r28)
    4844: 37 e3 00 00       addic. r31,r3,0
    4848: 40 82 00 90       bne  48d8 <MdlOutputs+0xbc>
    484c: 3c 60 00 05       lis    r3,5

smd_1005_s1.c:74
    /* Integrator: '<Root>/Integrator 2' */
    rtB.root_x = rtX.root_Integrator_2_CSTATE;
    /* SignalGenerator: '<Root>/x disp' */
    if (ssIsMajorTimeStep(rtS)) {
        4850: 3f 60 00 05       lis    r27,5
        4854: c8 23 e4 58       lfd    f1,-7080(r3)
        4858: d8 3b e4 68       stfd   f1,-7064(r27)
        485c: 3f a0 00 05       lis    r29,5
        4860: 80 64 01 3c       lwz    r3,316(r4)
        4864: 3f c0 00 05       lis    r30,5
        4868: 80 63 00 34       lwz    r3,52(r3)
        486c: 2c 03 00 01       cmpwi  r3,1
        4870: 40 82 00 58       bne  48c8 <MdlOutputs+0xac>
        4874: 3c 60 00 05       lis    r3,5
```

The `MdlOutputs` function is shown, and all machine operations executed within it and all associated addresses. In the `smd_1005_s1.c` file, you have to check the instruction given in line 74 if the error occurred at location 485c, for example.

5 Derive the respective source code line from the C source line close above the disassembled code. Check whether these instructions could have caused the exception.

6 If the specified function is associated with a Simulink block, you can find the block in your model and change its parameters.

See also How to Find the Simulink Block Related to an Exception on page 263.
How to Find the Simulink Block Related to an Exception

Real-Time Workshop writes system/block identification tags in the generated code. The tags are designed to help you identify the block in your source model that generated a given line of code. You should use the hilite_system command to trace a tag back to the generating block. For further information on identification tags and code tracing, refer to the Real-Time Workshop User's Guide by The MathWorks.

To find the Simulink block related to an exception

1. Investigate the model source code (<model>.c) for the Simulink block causing the problem: Real-Time Workshop begins the source code for each Simulink block with a token that indicates the block type and the actual block, for example,

```
/* Sum Block: <S1>/Sum */
rtB.s1_Sum = - rtB.s1_Product1 - rtB.s1_Product;
```

2. Make sure that the model is open, change to the MATLAB Command Window and enter the hilite_system('token') command. For the above-mentioned Sum block, this would read

```
hilite_system('<S1>/Sum')
```

Further information on the subsystem identifiers S1, S2, etc. is held at the end of the <model>.h file, for example:

```
/*
 * Comments are included throughout the generated code which allow you to trace directly back to the appropriate location in model. The basic format is <system>/block_name, where system is the system number (uniquely assigned by Simulink) and block_name is the name of the block.
 * Use the MATLAB hilite_system command to trace the generated code back to the model. For example,
 * hilite_system('<S3>') - opens system 3
 * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
```
Here is the system hierarchy for this model

* <Root> : smd_1005_sl
* <S1> : smd_1005_sl/Equation Block
* <S2> : smd_1005_sl/Model Parameters
* <S3> : smd_1005_sl/RTI Data

...
How to Create an Application for a Virtual dSPACE System

Suppose you want to create a real-time application for a dSPACE system that is not available at your developer PC and use another PC to download it to your dSPACE system. You have to use fictional board names and block numbers for this.

To create an application for a virtual dSPACE system

1. Create your model as normal using the usual board numbers.
2. Create the related custom C code as normal using the usual board names. For example, use DS5001_1_BASE to access the DS5001 board with the smallest PHS-bus address if you run a modular dSPACE system.
3. If you use RTI, build the model with the Load application after build option cleared (see Real-Time Workshop Page in the RTI and RTI-MP Implementation Reference).
   If you use RTI-MP, build the object files for all CPUs as normal, but do not download it.

To download an application that was created for a virtual dSPACE system

- Download the real-time application to the dSPACE system using ControlDesk's Platform Manager.

The board numbers of the application are mapped to the PHS-bus addresses of the I/O boards in ascending order (see PHS-Bus Address and Board Number on page 43).

If the real-time application does not match the available dSPACE system (for example, if the application references an I/O board that is not present), you get an appropriate error message.
Adding User-Specific Files, Source Files and Libraries

Suppose you want to add user-specific files, source files, and libraries to your application. RTI and RTI-MP provide the user makefile (USR.MK) for this purpose. It lets you modify the standard build process via a set of make macros. You can define extra search paths (for S-functions or user-defined C source code), user libraries, etc. For detailed information, refer to User Makefile (USR.MK File) in the RTI and RTI-MP Implementation Reference.
External Simulation

The external mode is a simulation mode provided by Simulink, Real-Time Workshop and RTI. This mode supports on-the-fly parameter tuning in a real-time environment. External refers to the fact that this mode involves two separate computing environments, a host and a target.

- The host is the computer that runs MATLAB and Simulink.
- The target is the dSPACE real-time hardware that executes the application you created with Real-Time Workshop/RTI.

Using external mode, you can modify the block parameters of the real-time application without regenerating and recompiling code, and without ControlDesk.

In external mode, parameter changes in the Simulink model are automatically downloaded to the real-time application that is currently running on your dSPACE hardware. The external mode of RTI and RTI-MP supports parameter tuning (downloading of parameters). It does not support data capture (uploading of the simulation results). For real-time data capture use ControlDesk.
Running an external simulation

Start and stop the external simulation and download parameters to the real-time simulation. Refer to How to Run an External Simulation on page 269.

Related Topics

Observing an external simulation

To observe an external simulation, you can use ControlDesk. Refer to the ControlDesk Experiment Guide.

Information provided by The MathWorks

For further information on external simulation, see Real-Time Workshop User’s Guide by The MathWorks.
How to Run an External Simulation

The following instructions describe how you can start the external simulation after building and downloading the real-time application and how you can change parameters of the real-time simulation via the Simulink model.

- If you run a multiprocessor system you cannot start the external simulation for the entire model as a whole. Instead, you have to start the external simulation individually for each CPU.
- External simulation can be activated only for boards residing in the connection that the current working board is in (bus/network). For example, if the working board is in a bus connection, external simulation cannot be performed on boards in a network connection.

To start the external simulation

1. Build and download your model as usual so that it is running on the dSPACE hardware.
2. If you run a multiprocessor system, open the submodel of the CPU from the MATLAB prompt (the submodels of all the CPUs are automatically generated during the build process). The name of a submodel is the same as the name of the corresponding application, which is listed in the 2nd column of the Application Setup frame on the Main page of the Multiprocessor Setup dialog.
4. Select the Simulation – Connect to target command.
To download parameters via the external simulation

1. Change the parameters in the Simulink [sub]model according to your requirements. By default, each parameter is downloaded immediately.

2. If you want to batch the download of several parameters, you can open the External Mode Control Panel dialog via the Tools – External Mode Control Panel command and select batch download.

- Selecting the batch download does not guarantee the data consistency of the downloaded parameters. Since all parameters are downloaded asynchronously to the execution of the model, the real-time simulation might run with an inconsistent parameter set for one or more sampling steps.
- The Arm trigger option should always be cleared, otherwise you get a warning that no blocks have been selected for uploading.

For information on the External Mode Control Panel, refer to External Mode Control Panel in the RTI and RTI-MP Implementation Reference.

To stop the external simulation

1. Select the Simulation – Disconnect from target command.

2. Before you can carry out a standard Simulink simulation for the model after working in the external simulation mode, select the Simulation – Normal command.
Conventions and Limitations

Even though RTI is designed to work seamlessly with Simulink and Real-Time Workshop, there are a few technically related conventions and limitations.

**Toolboxes and blocksets**

In addition to MATLAB and Simulink, The MathWorks offers a wide range of blocksets and toolboxes, which are extensions to the basic MATLAB and Simulink software packages. Refer to Supported Toolboxes and Blocksets from The MathWorks on page 272.

**Data typing**

MATLAB, Simulink, Real-Time Workshop, RTI, and RTI-MP support data typing. However, there are a few peculiarities. Refer to Data Typing Restrictions on page 273.

**Conventions**

There are a number of conventions for implementing models with RTI and RTI-MP. Refer to Conventions on page 274.

**Limitations**

Refer to Limitations on page 275.
Supported Toolboxes and Blocksets from The MathWorks

In addition to MATLAB and Simulink, The MathWorks offers a wide range of blocksets and toolboxes, which are extensions to the basic MATLAB and Simulink software packages.

As blocksets provide additional Simulink blocks, you must always know if these blocks can be used with Real-Time Workshop/RTI.

In most cases, toolboxes are simply extensions to MATLAB. This means that they generally do not provide additional Simulink blocks and that they have no impact on Real-Time Workshop/RTI compatibility. However, some toolboxes also provide additional Simulink blocks and you must know if these are supported by Real-Time Workshop/RTI too.

For blocksets and toolboxes from The MathWorks that provide Simulink blocks, the RTI compatibility is tested with selected demo models taken from the MATLAB installation and/or with test models designed especially for this purpose. These tests naturally cannot cover the complete functionality range of the blocksets. Although most possibilities are taken into account, the complexity resulting from the number of Simulink blocks and their possible parameterizations is far too large to guarantee tests with a 100% coverage.

As a result, problems might occur even for a blockset or toolbox that is stated to be supported by RTI. You can receive information about the known problems from blockset- or toolbox-specific pages at http://www.dspace.de/goto?Toolboxes or by contacting info@dspace.de.
Data Typing Restrictions

MATLAB, Simulink, Real-Time Workshop, RTI, and RTI-MP support data typing. However, there are a few peculiarities.

- If you are uncertain whether an integer signal might fall outside the valid signal range, you should select the **Saturate on integer overflow** checkbox in the Block Parameters dialog of the corresponding block. However, selecting this checkbox results in code that is computationally more intensive. If you use this option for a large number of integer blocks, the real-time code might become slower than if you were not using data typing.

- If you clear the **Saturate on integer overflow** checkbox for an integer block, an overflow leads to an unexpected result that depends on both the compiler and hardware.

- ControlDesk displays the data types according to their representation on the hardware; thus the data type displayed in ControlDesk's Variable Browser might differ from that defined in Simulink. This is not an error.

The following data types are used:

- All integer data types have the length specified in Simulink.
- The Boolean data type is represented by the unsigned 8-bit integer data type.
- The single data type is a 32-bit floating-point variable.
- The double data type is a 64-bit floating-point variable.
Conventions

There are a number of general conventions for implementing models with RTI and RTI-MP.

**Accessibility of parameters**  By default, ControlDesk can access only basic parameters of Simulink blocks, not additional parameters introduced in block masks. If you need to access mask or workspace parameters, you have to use the Include mask and workspace parameters option (see Real-Time Workshop Page (for RTI) and Variable Description File Options Page (for RTI-MP) in the RTI and RTI-MP Implementation Reference).

**File names**  Ensure that you give your Simulink model a different name from the files that were initially installed by dSPACE or MATLAB; for example, do not use the name ds1103.mdl. As a rule of thumb, you should not use file names that start with ds, rt or rti. This rule applies even if the file name extensions are different.

File and path names must not contain any blanks.

**Mask prompt names**  Do not use the " character (quotation mark) in mask prompt names if you want to include the mask and workspace parameters of the model in the variable description file. This character leads to syntactically incorrect variable description files since it is used as a string delimiter.

**Naming of interrupt blocks**  Do not use identical names for different interrupt blocks in your model. ControlDesk requires unique names for all of the Task Info's subgroups.

**Numerical integration**  For real-time simulation, use only fixed-step integration methods.

**Sampling time**  For discrete blocks located in a timer task, you have to set the sample times to integer multiples of the base sample time.

**Simulink block names**  Do not use the " (quotation mark) and / (slash) characters in Simulink block names. These characters lead to unusable variable description files.
Limitations

General
There are a number of general limitations for implementing models with RTI and RTI-MP. Refer to General Limitations on page 275.

RTI-MP
If you implement RTI-MP models, additional limitations apply. Refer to Limitations with RTI-MP on page 279.

Stateflow
If you include Stateflow® in your models, additional limitations apply. Refer to Limitations with Stateflow on page 280.

Custom C code
If you include custom C code in your models, additional limitations apply. Refer to Limitations for Custom C Code on page 280.

General Limitations

There are a number of general limitations for implementing models with RTI and RTI-MP.

Unsupported blocks
There are a few blocks which are not supported by Real-Time Workshop. For details, refer to Real-Time Workshop User’s Guide by The MathWorks. The most important unsupported blocks are:

- Algebraic Constraint block
- MATLAB Fcn block

Blocks depending on absolute time
Certain Simulink blocks have a C-code description that depends on the absolute simulation time (see Real-Time Workshop User’s Guide by The MathWorks for a list of the affected blocks). This leads to the following limitations:

- With RTI you can create models that contain no timer tasks but are driven only by interrupt blocks. As a result, the absolute simulation time remains 0 throughout the simulation. This makes the affected Simulink blocks nonfunctional.
Time-dependent blocks or S-functions fail to work correctly or deliver imprecise or even meaningless results if used in subsystems triggered by hardware interrupts. This is because the Real-Time Workshop-generated code for these blocks uses the absolute simulation time, which is updated with the fixed step size and can have a significantly reduced precision (as seen in tasks driven by hardware interrupt blocks).

Some Simulink blocks might be built up from Simulink blocks that depend on the absolute time. These resulting Simulink blocks are also time-dependent even if they are not explicitly listed in the Real-Time Workshop User’s Guide.

Nonfunctional signal sinks  Most blocks from the Simulink Sinks or Additional Sinks library have no function when simulated on dSPACE hardware. Instead, you can use ControlDesk. You may leave these blocks in your Simulink model – but keep in mind that RTI does not create C code for them. The affected signal sinks are shown below:

Simulation parameters  Fixed-step size “auto” is not supported.

Time-stamping  Currently the Real-Time Workshop-generated code does not support the time-stamping feature. Blocks that need the simulation time for their calculations use the currentTime variable instead.
The dSPACE Real-Time Kernel calculates the time stamps using integer numbers, whereas Real-Time Workshop calculates the `currentTime` within the fastest timer task using floating-point numbers. The increment of the `currentTime` is the fixed step size. Since the precision of floating-point numbers decreases the larger the numbers become, the `currentTime` variable and the associated time stamps might differ slightly.

**Time-stamping and non real-time simulation**  
If you do not use the “real-time” execution mode, you must turn off the time-stamping feature in ControlDesk. Otherwise, the x-axes of the plots in ControlDesk do not match the simulation time. This is because time stamps are derived from a real-time clock, which always delivers time stamps in real time regardless of whether the application is executed in real time or not.

**Task transition blocks**  
Task transition blocks have the following limitations:

- Task transition blocks support all data types except for complex ones.
- Task transition blocks do not support matrix signals. To transfer a matrix, use the Reshape block from the Simulink library, which converts the signal into a single vector. In the receiving task, you can reshape it back into a matrix.
- Task transition blocks do not support frame-based signals and there is no workaround available.
- Task transition blocks inherit their data types from their sources.
- After simulation start, if the receiving task is executed before the sending task, the output value of the task transition block is not defined. To specify an initial value to be used in this case, you can use Simulink’s IC (Initial Condition) block.

**Combining interrupts**  
In combining interrupts to trigger the same subsystem, the following limitations apply:

- The code generated for the function-call subsystem is not safe on reentrance. That is, the data of the first call of a task is lost if another call starts. This is due to the global definition of block outputs in the function-call subsystem. Therefore, if two tasks act simultaneously on the same function-call subsystem, you cannot
predict which block outputs will result. Depending on the blocks calculated in the task, even serious memory corruption can result. For this reason you should use identical priorities when combining interrupt sources. Tasks of the same priority do not interrupt each other.

- You cannot combine the function-call signals from RTI interrupt blocks with those of other blocks, for example, a Function-Call Generator block or Stateflow chart.

- You cannot use the `ssEnableSystemWithTid` and `ssDisableSystemWithTid` macros to enable and disable a function-call subsystem driven by an RTI interrupt block because the subsystem will always be executed when it is triggered by an RTI interrupt block.

**Code reuse feature** Due to a limitation of Real-Time Workshop, you cannot use the reusable function option for nonvirtual subsystems if you want to have mask and workspace parameters generated into the variable description file. In addition, reused subsystems are not included in the variable description file. Therefore, the dSPACE installation disables the code reuse feature for dSPACE RTI targets by default. This means that non-virtual subsystems are not generated as reusable functions regardless of the settings you specify for the subsystems of your models.

If you want to enable the code reuse feature (not recommended), carry out the following instructions:

**To enable the code reuse feature**

1. Make sure that mask and workspace parameters are not generated into the variable description file:
   - For RTI models, open Real-Time Workshop page of the Simulation Parameters dialog, change to the “RTI variable description file options” category and clear the **Include mask and workspace parameters** checkbox.
   - For RTI-MP models, open a CPU’s page of the Multiprocessor Setup dialog, then open the Variable Description File Options page of the CPU Options dialog and clear the **Include mask and workspace parameters** checkbox. This is a global option that affects all CPUs.
To enable the code reuse feature, change to the MATLAB Command Window and enter

\texttt{rti\_option EnableRTWCodeReuse 1}

The code reuse feature for non-virtual subsystems is then enabled for all models until you carry out the command:

\texttt{rti\_option EnableRTWCodeReuse 0}

**External simulation and interrupt blocks** If your model contains interrupt blocks, you cannot run an external simulation, because interrupt blocks inherently modify the model's checksum each time the model is initialized. However, the external simulation identifies the correct model for the real-time application via the checksum.

**Limitations with RTI-MP**

If you implement RTI-MP models, additional limitations apply.

**Mixed multiprocessor systems** RTI-MP does not support mixed multiprocessor systems that contain both DS1005 and DS1006 boards.

**Propagating labels over CPU boundaries** It is not possible to propagate label strings over CPU boundaries (across IPC blocks).

**Real-Time Workshop optimization** The Signal storage reuse and Inline Parameters options cannot be set in a CPU-specific way, only for the whole RTI-MP model (see Advanced Page in the RTI and RTI-MP Implementation Reference).

**Subscheduling only for local software interrupts** RTI-MP supports subscheduling only for local software interrupts. An interrupt is called local if its source CPU is the same as its target CPU.

**Combining interrupts** With RTI-MP you can combine only local interrupts to trigger the same subsystem, i.e., all interrupts must originate from the same CPU. None of the function-call signal lines may contain an IPI block.

**Transferring matrix signals via IPC block** 2-dimensional matrix signals cannot be transferred via an IPC Connector block. To do this, you can use the Simulink Reshape block to convert a matrix to a vector and vice versa.
Conventions and Limitations

Limitations with Stateflow

If you include Stateflow in your models, additional limitations apply.

**States of Stateflow charts** By default, the states of Stateflow charts are not accessible via the variable description file. If you need to trace the state activity for a statechart, you can use the **Output State Activity** option for the states you need to observe. The outputs of the individual Stateflow charts are available via the Model Root group. The states, parameters and local Stateflow chart data are not available. The State Machine Data group collects all data that is available in the Stateflow data dictionary and parented by the state machine.

For details on the Output State Activity option, refer to Stateflow by The MathWorks.

Limitations for Custom C Code

If you include custom C code in your models, additional limitations apply.

**Programming language** Both User-Code and S-functions must be written in C. RTI does not support C++.

**Limitations common to S-functions and User-Code**

**Access to internal variables** RTI generates all Simulink variables into the `<model>.trc` variable description file. You can access them during simulation, for example, via ControlDesk. However, RTI cannot include the internal variables of S-Function blocks and User-Code in this file. If you want to access them via ControlDesk, you have to declare them as global and create a user variable description file (`<model>_usr.trc`). Refer to User Variable Description File (USR.TRC File) in the RTI and RTI-MP Implementation Reference.

**I/O function calls** There are no error checks regarding the correct use of the I/O function calls; for example, to check if the same I/O channel is accessed twice.

**Unsupported C statements** Certain C statements are not supported for real-time applications, for example, console I/O, file I/O, etc.
Limitations concerning S-functions

**M-coded S-functions**  Do not use any M-coded S-function for real-time applications, as the C compilers always need C code. Keep in mind that even some of the original Simulink blocks use M-coded S-functions.

**Reuse of S-functions**  If your model contains more than one instance of a specific S-function that uses global variables, unexpected side-effects in the application may occur.

**Generated S-function**  With Real-Time Workshop, it is possible to generate code from a Simulink model and then hook the generated C code to the RTW S-Function block. As a result you can reduce the build times because the code contained in the RTW S-Function block does not need to be regenerated or recompiled. Such an S-function cannot contain any RTI- or RTI-MP-specific block (for example, I/O).

After you have built a model using the RTW S-Function, Real-Time Workshop opens a new model and inserts a correctly configured RTW S-Function block.

Even though the generated S-function is derived from a Simulink model, it is not possible to access its internal variables via the variable description file. You will only be able to see the block outputs and parameters of the S-function.


Limitations concerning User-Code

**User-Code driven by an interrupt block**  You cannot use the `<model>_usr.c` User-Code file to add custom code to the tasks driven by interrupt blocks. Instead, use an S-function or Real-Time Workshop's Custom Code block in a subsystem that is triggered by the desired interrupt block.
Troubleshooting

When implementing an RTI or RTI-MP model, when building and downloading the real-time application or when running the simulation, you might experience problems. If any RTI-related problem occurs, follow the instructions provided by the error message.

The error messages are listed in alphabetical order. Problems without a message are placed first. Terms in angle brackets <...> refer to specific parts of your model, for example, a specific block, subsystem or CPU.

**Implementation phase**

When you implement your model and encounter problems or error messages, refer to Implementing the Model on page 285.

**Build and download**

When you start automatic program building and encounter problems or error messages, refer to Automatic Program Building on page 289.

If this information does not help you solve the problem, you should check the support section of our Web site http://www.dspace.de/goto?support. This might help you solve the problem on your own. The support's FAQ section especially might be of help.
If you cannot solve the problem, contact dSPACE Support via dSPACE Support Wizard. It is available:

- On your dSPACE CD at `\Diag\Tools\dSPACESupportWizard.exe`
- Via `Start – Programs – dSPACE Tools` (after installation of the dSPACE software)
- At `http://www.dspace.de/goto?supportwizard`

You can always find the latest version of dSPACE Support Wizard here.
Implementing the Model

These problems and error messages might occur when you implement your Simulink model.

RTI-MP error message  
**<block> is not a valid Interrupt block for IPI block <name>**
An IPI block is always required between a hardware or software interrupt block and the corresponding function-call subsystem. The trigger type of this subsystem must be set to function-call.

➤ Ensure that an IPI block is included in each connection between the hardware or software interrupt block and its corresponding function-call subsystem.

RTI-MP error message  
**<subsys> is not a triggered subsystem**
An IPI block is always required between a hardware or software interrupt block and the corresponding function-call subsystem. The trigger type of this subsystem must be set to function-call.

➤ Ensure that an IPI block is included in each connection between the hardware or software interrupt block and its corresponding function-call subsystem.

RTI-MP error message  
**All integration algorithms and step sizes must be equal if central integration is used**
For the central integration algorithms (ode2-mp, ode3-mp, ode4-mp, or ode5-mp), using different step sizes or integration algorithms on the different CPUs of your model is not allowed.

➤ If you want to use a central integration algorithm, you have to ensure that all CPUs are configured with the same solver and integration step size.

➤ If you want to run the different CPUs with different solver types or integration step sizes, you have to ensure that you do not use a central integration algorithm for any CPU.
**Troubleshooting**

**RTI Block displays**  
**Bad Link**  
This appears if a block comes from an old RTI or an RTI that is different from the one you are currently using.

The RTI library uses Simulink library links. If you include blocks from the RTI library in your application, they function as references to the original blocks of the library. Therefore, any library updates automatically update your application as well. If the changes to a new block are too significant in comparison to the block you are using in your application, this message is shown.

➤ Replace the old block manually with the new one from the library.

**RTI-MP error message**  
**Error in MP network topology!**  
CPU `<cpu>` is not a member of the registered MP system.  
The following CPUs are registered with ControlDesk or have been specified in the MP Topology setup  
`<...>`

The Simulink multiprocessor model contains one or more CPUs that have not been specified in the Multiprocessor Topology Setup dialog.

➤ Rename the CPU `<cpu>` in the Multiprocessor Setup dialog or adapt the settings in the Multiprocessor Topology Setup dialog.

➤ Change either the GigaLink connections of your hardware according to your Simulink model or redesign your Simulink model so that it matches the network topology of your hardware.

**RTI-MP error message**  
**Hardware configuration**  
Allocation of `<swinging buffer | virtual shared memory>` connection from `<cpu1>` to `<cpu2>` failed.

The number of available swinging buffer channels/virtual shared memory channels is not sufficient for the model configuration (each GigaLink connection provides eight swinging buffer channels and seven virtual shared memory channels per direction).

➤ Change your Simulink model so that it does not use more than eight swinging buffer channels and seven virtual shared memory channels between `<cpu1>` and `<cpu2>` (per GigaLink connection and direction). As an alternative, you can add further GigaLink connections to your hardware to match the requirements of your Simulink model.
Troubleshooting

RTI-MP error message  Hardware configuration
Missing gigalink connection from <cpu1> to <cpu2>.
An IPC channel has been specified between <cpu1> and <cpu2>, but the CPUs are not directly connected via GigaLinks.
➤ Install all the required GigaLink connections.

RTI-MP error message  Incompatible Step Size Multiples
The destination Step Size Multiple <n> of connection <cpu1 to cpu2 on ch2> must be a multiple of the Step Size Multiple <m> of destination CPU <cpu2>!
All communication channels require that their source and destination step sizes are compatible with the step size of the corresponding source and destination CPUs.
➤ For each communication channel of your model ensure that its source and destination step sizes are compatible with the step sizes of the corresponding CPUs. All of these step sizes have to be entered as integer multiples of the multiprocessor system's basic step size. You have to check that:
   • The source step size of the channel is an integer multiple of the source CPU's step size,
   • The destination step size of the channel is an integer multiple of the destination CPU's step size.

RTI-MP error message IPC Connector block <name> must not be located in block diagram-depth>2
IPC Connector block <name> must not be located in subsystem <subsys>
All IPC blocks must be placed either on the root level of the model or on the top level of a subsystem that is driven by an interrupt block or assigned to the background task.
➤ Place the IPC blocks of your model only on the root level of the model or on the top level of triggered subsystems.
RTI-MP error message  No Interrupt block found for IPI Block <name>  

No triggered subsystem found for IPI Block <name>  

An IPI block is always required between a hardware or software interrupt block and the corresponding function-call subsystem. The trigger type of this subsystem must be set to function-call.

➤ Ensure that an IPI block is included in each connection between the hardware or software interrupt block and its corresponding function-call subsystem.

RTI-MP error message  Source and Destination CPU of line between blocks <blockX> and <blockY> are not equal  

All blocks of the multiprocessor model inherit their CPU identity from the IPC blocks they are connected to. If your model contains a connection between blocks that are assigned to different CPUs, their CPU identity is ambiguous.

➤ Ensure that the CPU identity of all of the blocks in your model is unambiguous.

RTI-MP error message  Incompatible Step Size Multiples  
The source Step Size Multiple <n> of connection <cpu1 to cpu2 on ch2> must be a multiple of the Step Size Multiple <m> of source CPU <cpu1>!

All communication channels require that their source and destination step sizes are compatible with the step sizes of the corresponding source and destination CPUs.

➤ For each communication channel of your model ensure that its source and destination step sizes are compatible with the step sizes of the corresponding CPUs. All of these step sizes have to be entered as integer multiples of the multiprocessor system's basic step size. You have to check that:

• the source step size of the channel is an integer multiple of the source CPU's step size,
• the destination step size of the channel is an integer multiple of the destination CPU's step size.
Automatic Program Building

During automatic program building, RTI uses a number of tools that might issue error messages. Most of these error messages are self-explanatory. Therefore you will find instructions only for:

- Error situations that might be misleading or do not always point to the reason for the problem,
- Irregular program behavior.

The instructions are structured according to the steps RTI carries out when you initiate the process of building and downloading real-time code.

The following screen output of MATLAB is taken from a successful, error-free build of a model (model.mdl) for a DS1005. Thus, when you build a model and encounter a problem, you can find the step where the problem occurred by comparing your screen output with the corresponding section of this example. RTI-MP's build process for a single submodel is very similar (the lines enclosed in brackets \[\]) refer to building a model for a multiprocessor system). RTI-MP does not download the MP application immediately.

**Model check**

Starting build procedure with RTI 5.0 (RTI<xxxx>, <DATE>)

Model: "<model>" (F:\dSPACE_RLS40\Demos\...\simulink\<model>.mdl)

*** Working directory: "F:\dSPACE_RLS40\Demos\...\simulink"

Refer to Checking the Model on page 291.

**File generation**

[*** Generating communication functions for application "<submodel>" ...]

[*** Generating real-time simulation engine for application "<submodel>" ...]

*** Optional User System Description File <model>_usr.sdf not available

*** Initializing code generation

### Starting Real-Time Workshop build procedure for model: <model>

### Generating code into build directory: .\<model>_rti<xxxx>

### Invoking Target Language Compiler on <model>.rtw

*** Generating Variable Description File <model>.trc

*** Optional User Variable Description File <model>_usr.trc not available

*** Generating template: User-Code File <model>_usr.c

*** Generating template: User Makefile <model>_usr.mk

**Compiler, linker, loader**

### Building <model>: dsmake -f <model>.mk WORKINGBOARD=ds<xxxx>

BUILDING APPLICATION (Single Timer Task Mode)
WORK DIRECTORY "F:\dSPACE_RLS40\Demos\...\simulink"  
BUILD DIRECTORY "F:\dSPACE_RLS40\Demos\...\simulink\<model>_rti<xxxx>"  
TARGET COMPILER "C:\<compiler_path>"  
COMPILING <model>.c  
[...]  
USING LIBRARY "D:\dSPACE4.2\MATLAB\RTI<xxxx>\C\Lib\rtwlib_r13_1_ds<xxxx>.lib"  
LINKING APPLICATION ...  
LINKING FINISHED  
LOADING APPLICATION "<model>.sdf" ...  
[...]  
LOADING FINISHED  
MAKE PROCESS SUCCEEDED  

Refer to Compiling and Linking the Code on page 294.

### Successful completion of Real-Time Workshop build procedure for model: <model>  
*** Finished RTI build procedure for model <model>  

Refer to:  
- Initializing the Application on page 296  
- Running the Application on page 297  
- Running ControlDesk on page 300
Checking the Model

The following errors might occur when RTI checks your Simulink model. They appear either in the MATLAB Command Window or in a MATLAB or Simulink message box.

Error messages

All sources of function-call subsystem block `<SubSystemName>`, input port `<PortNumber>` must be valid rate transition blocks such as a double buffer block or a rate transition block.

Function-call subsystem block `<SubSystemName>`, output port `<PortNumber>` must have only one destination block, which must be a valid rate transition block such as a double buffer read block or a rate transition block.

Invalid connection for asynchronous rate transition block `<BufferedTaskTransBlockName>`. Either the source or destination block must be asynchronous. If the destination is an asynchronous subsystem, then the rate transition block cannot have any other destinations.

Your model contains one or more direct signal lines between an interrupt block-driven task and another task, which is not allowed.

- Implement valid rate transitions by means of the Non-Buffered Task Transition block or the Buffered Task Transition (Write) block and the Buffered Task Transition (Read) block, which you can find in RTI's TaskLib.

Refer to Preparing a Function-Call Subsystem on page 78.

Do not use the blocks from the VxWorks library of Real-Time Workshop (for example, the Rate Transition block) because these are not supported by RTI. Instead, use the task transition blocks from RTI's TaskLib.
Error message **Cannot find product license**
The RTI or RTI-MP license entry in the license.dsp license file is missing.
➤ Check your license file to find a valid product entry. If there is no license for the needed product, contact dSPACE to get the needed license.

Error message **Cannot find the license file**
The %DSPACE_ROOT%\Config\license.dsp file was not found. This contains the license codes for all dSPACE products.
➤ Check the setting of the DSPACE_CONFIG environment variables and whether the license.dsp file exists in the above-mentioned folder.

Error message **Execution key invalid or not found**
The execution key (dongle) cannot be accessed. Most probably the execution key is not plugged into your PC at all or plugged onto a wrong port.
➤ Ensure that the execution key is plugged in correctly.

Error message **RTI Build Error**
Incompatible Real-Time Workshop VxWorks blocks. The model employs blocks from the Real-Time Workshop VxWorks Library. These blocks are not designed for use with RTI.
-> <VxWorksBlockName>
Replace them with blocks from the RTI TaskLib.
Your model contains blocks from the VxWorks library of Real-Time Workshop. These are not supported by RTI.
➤ Implement valid rate transitions by means of the Non-Buffered Task Transition block or the Buffered Task Transition (Write) block and the Buffered Task Transition (Read) block, which you can find in RTI's TaskLib.
Refer to **Preparing a Function-Call Subsystem** on page 78.
Troubleshooting

Error message

**The product license is not valid**

The license strings of RTI or RTI-MP probably belong to an earlier version of RTI or RTI-MP. Maybe the license entry does not match the serial number of the execution key (dongle) connected to the PC.

➤ Check your license file to find a valid product entry. If there is no license for the needed product, contact dSPACE to get the needed license.

RTI-MP warning

**Warning: Unable to honor user-specified block priorities. Block `<block1>` (pri=10000) has to execute before block `<block2>` (pri=-10000) to satisfy data dependencies.**

Either the IPC blocks `<block1>` and `<block2>` form a direct feedthrough or you specified illogical block priorities. As a result, the block priorities are ignored, and the IPC Input block is calculated before the IPC Output block.

➤ Try to avoid direct feedthrough. You should also avoid specifying block priorities manually. See Model Evaluation on page 133 for details.

**Generating the Code**

The following errors might occur while RTI generates real-time code from your Simulink model. They appear either in the MATLAB Command Window or in a MATLAB or Simulink message box.

Error message

**Error using ==> rtwgen**

---> Invalid “function-call” connection

(The message might be longer.)

The Trigger block in a subsystem triggered by a hardware or software interrupt is not set to the function-call type.

The Trigger block in a subsystem triggered by a Simulink signal is set to the function-call type.

1 Set the **Trigger** blocks of subsystems triggered by hardware or software interrupts to the **function-call** type.

2 Set the **Trigger** blocks of subsystems triggered by Simulink signals to one of the following types: **rising**, **falling**, or **either**.
Compiling and Linking the Code

The following errors might occur while RTI compiles and links the real-time code. They appear in either the MATLAB or the DOS window.

**Caution: Current program unit “MdlOutputs” is too large for constant and copy propagation optimizations, turning them ‘off’**

**Caution: Main Global Optimizations not done, recompile with -OXM<nnn> (Mbytes)**

**Caution: Main Global Optimizations not done, recompile with -Xmemlimit=<nnn> (Mbytes)**

During the compilation of a large model, the Microtec PowerPC Compiler issues a warning. Nevertheless, the compilation is finished and a valid application object created. Therefore, you can ignore this warning message unless you run into execution speed problems (task overruns) of the real-time application.

This warning is issued by the PPC Optimizer and means that additional host PC memory is necessary for full optimization.

1. For single-processor systems, open the Simulation Parameters dialog of the model, change to the Real-Time Workshop page and select the “RTI general build options” category.

   For multiprocessor systems, open the Multiprocessor Setup dialog of the model, change to the individual CPUs’ pages and open the CPU Options dialog on the Build Options page.

2. Enter `-OXM<nnn>` at the Compiler options setting, using the number `<nnn>` as given in the warning message, for example, `-OXM399`. 
Error message  "model_usr.c", line 39: [F108] ‘u’ undefined
(or similar)
The User-Code of your model contains pointer variables that access
root-level Inport or Outport blocks. Due to internal Real-Time
Workshop changes imposed by data typing, this feature is no longer
supported by RTI.
➤ Use Data Store blocks as described in How to Use User-Code in
Simulink Models on page 164 instead of pointer variables.

Error message  OPUS MAKE aborts the build process
If OPUS MAKE cannot finish the build process successfully, this is most
often due to missing files, incorrect installation of the compilers, or of
the dSPACE software.
➤ Check if the following environment variables exist and point to the
correct directories where the compilers and dSPACE software are
installed:
• $DSPACE_ROOT%
• $PPC_ROOT%
• $X86_ROOT%

Error message  OPUS MAKE: Don't know how to make <sfname>.<…>. Stop
(This appears while the make utility is running.)
Your model contains user-written S-functions, but the make utility
cannot find the corresponding C-coded source file.
Your model contains Simulink blocks consisting of M-coded
S-functions.
1 Place the C-coded source files of your S-functions in the same
working folder as the model, or specify the S-function folder in the
user makefile. To specify the S-function folder in the user makefile,
add the name of the folder to the entry
# Directories for S-Functions
SFUN_DIR =
You can find the user makefile template <model>_usr.mk in the
working folder of the model.
2 Do not use M-coded S-functions in Simulink models that you want
to use for building real-time programs.
Initializing the Application

The following errors might appear in ControlDesk's log file or Log Viewer.

If an error occurs at this stage, RTI exits the application. Hence, both the real-time application and the background task terminate. The error might result from hardware or from the initialization of the User-Code or an S-function. You cannot restart your application, but have to download it again after you have solved the problem.

Error message **Application terminated**

(The application stopped executing right after downloading; meaning that the real-time application might not have finished its initialization correctly.)

During the initialization of an S-function or User-Code an `exit()` command was encountered.

Due to programming errors, the real-time program crashed during the initialization phase.

➤ Check the S-functions and User-Code of the model for `exit()` commands.

➤ Try to run your application without S-functions and User-Code to detect whether possible programming errors might be causing the problem. If this is the case, fix these programming errors.

Error message **Memory allocation failed**

(Or a similar message.)

The real-time processor has run out of dynamic memory. This situation might occur when you use S-functions that allocate large amounts of dynamic memory.

➤ Check if your Simulink model uses such S-functions. If this is the case, try to reduce the amount of the allocated dynamic memory.
Running the Application

The following errors and messages might occur while the dSPACE system runs your real-time application. These errors and messages appear in the ControlDesk's log file or Log Viewer.

If your application stops executing, check whether there are any messages. If an error occurs in this stage, RTI switches the simulation to the STOP mode. Hence, the real-time application terminates, but the background task is still active. The error might result from your application or from the `usr_output()`, `usr_sample_input()` or `usr_input()` sections of your User-Code or S-function.

No message

[-]
(The application has stopped.)

The real-time program hung due to a statement that corrupted the real-time program in an S-function or User-Code of your model.

➤ If your model uses S-functions or User-Code, try to run it without these parts to detect whether programming errors might be causing the problem.

Exception message

Alignment Error (PowerPC processor)
(The application has stopped.)

The processor can read or write only aligned data. 64-bit data types can be accessed only at addresses on an 8-byte boundary (ending with 0H or 8H), 32 bit data types at addresses on a 4-byte boundary (ending with 0H, 4H, 8H, CH). If an attempt is made to access data at other addresses, this exception is raised.

➤ Debug your application and find the block or function that is responsible for this exception.

Refer to How to Trace an Exception on page 259.
Exception message  **Data Access Error (PowerPC processor)**
(The application has stopped.)
The processor cannot read data from or write data to the specified address. An attempt was made to access a memory location where there is physically no memory present.
➤ Debug your application and find the block or function that is responsible for this exception.
Refer to *How to Trace an Exception* on page 259.

Exception message  **Instruction Access Error (PowerPC processor)**
(The application has stopped.)
The processor cannot fetch the instruction at the specified address. An attempt was made to access a memory location where memory is either not present or protected.
➤ Debug your application and find the block or function that is responsible for this exception.
Refer to *How to Trace an Exception* on page 259.

Exception message  **page fault exception (x86 processor)**
(The application has stopped.)
The processor cannot access the specified address. An attempt was made to access a memory location where memory is either not present or protected.
➤ Debug your application and find the block or function that is responsible for this exception.
Refer to *How to Trace an Exception* on page 259.

Message  **Model has requested to stop simulation**
(The application has stopped.)
The Stop block has stopped the simulation due to an input signal other than zero.
➤ Check the signal source for the Stop block.
RTI-MP error message  

**Num. of cont. states must be nonzero for central integration**  
If a central integration algorithm is selected (odeX-mp), the integration algorithms must be the same on all CPUs. The number of continuous states must be nonzero for each application, otherwise Real-Time Workshop automatically uses the dummy integration algorithm of type discrete as the solver. This would cause overload errors, deadlocks, or unexpected system behavior.  
➤ Do not use a central integration algorithm if there are applications in your multiprocessor model which do not have any continuous states.

Exception message  

**Program Error (PowerPC processor)**  
(The application has stopped.)  
This exception is raised if one or more of the exception conditions defined for the PowerPC architecture occur.  
The exception conditions are:  
■ Illegal instruction  
  This exception occurs when the execution of an instruction with an illegal opcode is attempted.  
■ Privileged instruction  
  This exception occurs when the execution of a privileged instruction is attempted and the MSR register user privilege bit is set.  
■ Trap  
  This exception occurs when any of the conditions specified in a trap instruction is met.  
➤ Debug your application and find the block or function that is responsible for this exception.  
Refer to How to Trace an Exception on page 259.

Message  

**Simulation Stop time reached**  
(The application has stopped.)  
The stop time for the simulation has been reached.  
➤ Specify a larger value or \texttt{inf} for the stop time.
**Troubleshooting**

<table>
<thead>
<tr>
<th>Error message</th>
<th>Task Overrun: Occurred in task &quot;&lt;task name&gt;&quot;, execution step &lt;number&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(The application has stopped.)</td>
</tr>
<tr>
<td></td>
<td>An overrun situation has occurred, because the scheduler attempted</td>
</tr>
<tr>
<td></td>
<td>to start a task that was still executing. This might result from a sample</td>
</tr>
<tr>
<td></td>
<td>time that was too small.</td>
</tr>
<tr>
<td></td>
<td>➤ Choose either bigger sample times for your timer tasks or a bigger</td>
</tr>
<tr>
<td></td>
<td>base sample time.</td>
</tr>
<tr>
<td></td>
<td>➤ Choose a different execution mode (single timer task mode or</td>
</tr>
<tr>
<td></td>
<td>multiple timer task mode) for your model.</td>
</tr>
<tr>
<td></td>
<td>➤ Try a different distribution of priorities over your tasks.</td>
</tr>
<tr>
<td></td>
<td>➤ Try a different overrun behavior.</td>
</tr>
<tr>
<td></td>
<td>➤ If you still encounter task overruns, you can use a more powerful</td>
</tr>
<tr>
<td></td>
<td>dSPACE board, or migrate the model to a multiprocessor system. If</td>
</tr>
<tr>
<td></td>
<td>you already use a multiprocessor system, you can enlarge it.</td>
</tr>
</tbody>
</table>

**Running ControlDesk**

The following messages might occur while you are working with your real-time application. These messages are issued by ControlDesk and can be found in ControlDesk’s log file.

<table>
<thead>
<tr>
<th>No message</th>
<th>[...]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(The update of ControlDesk's virtual instruments is very slow.)</td>
</tr>
<tr>
<td></td>
<td>The different tasks of the real-time program are using almost 100% of</td>
</tr>
<tr>
<td></td>
<td>the processor's capacity.</td>
</tr>
<tr>
<td></td>
<td>The Simulink model contains a Background block connected to a</td>
</tr>
<tr>
<td></td>
<td>subsystem that is computationally intensive.</td>
</tr>
<tr>
<td></td>
<td>➤ Free more processor time for the background task by increasing</td>
</tr>
<tr>
<td></td>
<td>the step size of the model.</td>
</tr>
<tr>
<td></td>
<td>➤ Use the Background block only for subsystems that are not</td>
</tr>
<tr>
<td></td>
<td>computationally intensive.</td>
</tr>
</tbody>
</table>
No message

[...] 
(There are no block parameters available below the model root node in the Variable Browser of ControlDesk.)
You selected the Inline parameters checkbox on the Advanced page of the Simulation Parameters dialog. When parameters are inlined, Real-Time Workshop does not allocate variables for them but uses constants instead.
➤ Clear this checkbox and rebuild the model.
➤ Use the Tunable Parameters feature.

Error message

Cannot find MAP file
(This appears when you attempt to load the <model>.sdf file in ControlDesk.)
There is no <model>.map file available because the linker could not build the <model>.obj file.
➤ Build the model again to get a <model>.map file.

Message

Trace capture in progress
(This appears after you started the data capture, but the data acquisition does not continue.)
The real-time application has terminated (STOP or PAUSE mode).
The host service code is located in a task that is currently idle.
➤ Use ControlDesk to switch the simulation into RUN mode via the simState variable.
➤ Either enable the execution of the task containing the Data Capture block, or place the block into another task.
Symbols

<model> 13
<submodel> 13

A
absolute time 275
accessing RTI blocks 30
acquiring data 200
activating platform support 29
adding communication 129
custom C code 153
default CPUs 145
I/O blocks 45
multiprocessor support 117
ports
User-Code 169
task handling 51
addresses
PHS-bus 43
S-functions 162
algebraic loops 137
AlphaCombo
migrating to DS1005 234
migrating to DS1006 234
applications
building 241
debugging 256
errors in custom C code 186
assertion handling 238
asynchronous tasks 75
atomic subsystems 214
reducing file sizes 216
function sizes 216
automating RTI-MP builds 250

B
background tasks 93
blocks bundling 214
execution order 223
model verification 238
multiplexed I/O 47
names 41
priorities 225
scalar I/O 47
board libraries 31
board numbers 41
buffering data 138
building
default options 34
models 241
optimizing 213
overview 244
rebuilding
SDF files 253
TRC files 253
bundling blocks 214

C
C code 153
comparing 174
S-functions 158
C methods 159
MATLAB_MEX_FILE 160
modular hardware 162
rebuilding 251
User-Code 168
adding ports 169
interprocessor communication 171
variables in ControlDesk 184
capturing
data 205
channels
communication 123
I/O block 41
code format 243
combining interrupt sources 101
commands
distartup.m 33
rti_build 250
rti_mdlcleanup 38
rti_sdfmerge 253
rti_usrtrcmerge 253
rtimp_build 250
rtimpdiag 256
set_rti 37
startup.m 33
communication 123
adding 129
channels 123
configuring 129
connections
Gigalink 124
sample time transitions 137
compiling models 241

configuring communication 129
RTI-MP models 120
connections
Gigalink 124
task design 18
conventions 274
I/O 41
converter numbers 41
CPUs
default 119
master 120
optional 119
currentTime
background task 63
precision problems 202
time-stamping 276
custom C code 153
execution order 175
rti_build 251
rtimp_build 251
variables in ControlDesk 184
customizing MATLAB start-up 33

D
data
acquisition 200
modifying 205
buffering 138
consistency
tasks driven by interrupt blocks 78
timer tasks 70
typing 49
enabling 50
restrictions 273
unbuffering 138
deadlock situations 137
debugging 256
default build options 34
CPUs 145
development steps 18
diagnostic tool 256
direct feedthrough 135
downloading
DS230c<0> applications 254
models 241
DS1005
migrating from AlphaCombo 234
DS1006
migrating from AlphaCombo 234
dSPACE HelpDesk 10
distartup.m 33

303
Index

E
- enabled subsystems 214
- enabling
  - data typing 50
  - errors
    - handling 186
    - reaction on run-time errors 208
- examples
  - FFT 69
  - RTI-MP 144
  - I/O block names 42
  - throttle control 20
- exceptions 258
- Excluding
  - subsystems 218
- execution order
  - blocks 223
  - bundling 214
  - custom C code 175
  - priorities 96
  - RTI-MP 149
  - tasks 58
- external simulation 267

F
- feedthrough
  - direct 135
- FFT example 69
- RTI-MP 144
- files
  - reduced size
    - TRC file 218
    - reducing sizes 216
  - functions
    - reducing sizes 216

G
- GigaLink connections 124
- group numbers 41

H
- handcrafting
  - C-coded S-functions 158
  - User-Code 168
- handling 258
- assertions 238
- errors in custom C code 186
- exceptions 258
- tasks 51
- hardware interrupt blocks 87
- HelpDesk 10

I
- I/O blocks 47
  - adding 45
  - conventions 41
  - unused channels 46
  - idle 58
  - inlined S-functions 157
  - interprocessor communication 123
    - channels 123
    - sample time transitions 137
    - User-Code 171
  - interprocessor interrupts 147
  - interrupt numbers 41
  - interrupts
    - asynchronous 75
    - interprocessor 147
    - tasks driven by interrupt blocks 75
      - combining 101
      - data consistency 82
      - hardware 87
      - software 89
      - subscheduling 104
  - timer tasks 63

K
- kernel 55

L
- libraries
  - board 31
  - limitations 275
    - custom C code 280
    - data typing 273
    - RTI-MP 279
  - Stateflow 280

M
- makefile
  - user makefile 266
- master CPU 120
- MATLAB
  - customizing start-up 33
  - MATLAB_MEX_FILE 160
- Message Module 186
- migrating from AlphaCombo to DS1006 234
- migrating models
  - with RTI 37
  - without RTI 38
- model verification blocks 238
- models
  - building 241
  - downloading 241
  - evaluation 133
  - multiprocessor distribution 151
  - modular hardware
    - S-functions 162
    - module
      - numbers 41
      - types 41
  - multiple timer task mode 66
  - data consistency 70
  - multiplexed I/O blocks 47
  - multiprocessor systems 117
  - multitasking 68

N
- naming conventions 41
- network topology 122
- noninlined S-functions 157
- numbers
  - communication channels 131
  - production channels 131

O
- optimizing
  - build process 213
  - RTI-MP model distribution 151
  - simulation 213
  - options 245
    - default 34
  - overrun situations 60
  - overview
    - build processes 244
    - development steps 18
    - execution order
      - blocks 223
      - custom C code 175
      - RTI-MP 149
      - tasks 58
    - naming conventions 41
    - PHB-bus address 43
  - RTI and RTI-MP Guide 24

P
- parameter tuning
  - State-Space blocks 226
  - Transfer Fcn blocks 226
  - Zero-Pole blocks 226
PDF files 11
PHS-bus address 43
platform support 29
port numbers 41
precision problems 202
preempting tasks 67
printed documents 11
priorities 56
blocks 223
tasks 96
RTI-MP 149
program building 241
protocols
swinging buffer 125
virtual shared memory 128

R
rate-monotonic scheduling 55
RCP 18
ready 58
real-time kernel 55
rebuilding
SDF files 253
S-functions 251
TRC files 253
reducing
file sizes 216
function sizes 216
restrictions 275
custom C code 280
data typing 273
RTI-MP 279
Stateflow 280
rtbsXXXX 12
rti_build 250
rti_mdlcleanup 38
rti_sdfmerge 253
rti_usrtrcmerge 253
rtiAssertionMode 239
RTI-MP
automating builds 250
communication
channels 123
sample time transitions 137
configuring models 120
diagnostic tool 256
execution order 149
network topology 122
rtimp_build 250
rtimpdiag 256
RUN/STOP mechanism 195
running 58
run-time errors 208

S
sample times
offsets 67
transitions 70
scalar I/O blocks 47
scheduler 55
selecting
platform support 29
set_rti 37
S-functions 156
C methods 159
MATLAB_MEX_FILE 160
modular hardware 162
rebuilding 251
shared memory 128
simState 197
simulation
control 195
external 267
optimization 213
start/stop 195
state 197
time 276
single tasking 68
single timer task mode 65
slave applications 188
software interrupt blocks 89
start/stop mechanism 195
starting
RTI 27
simulation 195
start-up
customizing MATLAB 33
state
simState 195
simulation 195
Stateflow
limitations 280
stopping
simulation 195
subscheduling
tasks 104
subsystems
atomic 214
enabled 214
excluding 218
trigger types 76
triggered 214
virtual 213
swinging buffer protocol 125
switching time 56
systems
multiprocessor 117
virtual hardware 265

T
tasks 51
asynchronous 75
background 93
combining interrupts 101
driven by interrupt blocks 75
exececution order 58
RTI-MP 149
multiple timer task mode 66
data consistency 70
multitasking 68
preempting 67
principles 52
priorities 96
single tasking 68
single timer task mode 65
states 58
subscheduling 104
switching time 56
timer 63
timetable 106
time-triggered 106
transitions 78
templates
RTI-MP standard topology 122
User-Code 167
throttle control example 20
time
absolute 275
stamping 200
timetable 106
triggered 106
timer
interrupt blocks 89
tasks 63
data consistency 70
timetable task 106
time-triggered task 106
topology
RTI-MP system 122
TRC file
reducing size 218
triggered subsystems 214
troubleshooting 283
tuning parameters
State-Space blocks 226
Transfer Fcn blocks 226
Zero-Pole blocks 226
turnaround times 60
types
data 49
triggered subsystems 76
Index

U
unbuffering data 138
unused I/O channels 46
user makefile 266
User-Code 164
  adding ports 169
  in Simulink 164
  interprocessor communication 171
templates 167

V
variables
  currentTime
    background task 63
    precision problems 202
    time-stamping 276
custom C code 184
  rtiAssertionMode 239
  simState 197
vectorized I/O blocks 47
virtual
dSPACE systems 265
subsystems 213
virtual shared memory protocol 128

W
wrapper S-functions 157