Embedded Target for Infineon C166® Microcontrollers

For Use with Real-Time Workshop®

Modeling

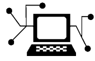
Simulation

Implementation

User's Guide

Version 1

How to Contact The MathWorks:

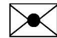


`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 Phone

508-647-7001 Fax



The MathWorks, Inc. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

Embedded Target for Infineon C166 Microcontrollers User's Guide

© COPYRIGHT 2002 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

Infineon, C166, and MiniMon are registered trademarks or salesmarks of Infineon AG.

Tasking is a registered trademark of Altium Limited.

PHYTEC is a trademark of Phytec Technologie Holding AG.

ST10 is a trademark of the STMicroelectronics Group.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History: November 2002 Online only Version 1.0 (Release 13+)

Contents

Preface

Installing the Embedded Target for Infineon C166® Microcontrollers vi

Using This Guide vii

Required Products viii
Related Products viii

Typographical Conventions x

Demos

Embedded Target for Infineon C166® Microcontrollers Demos xii

1 Product Overview

Prerequisites 1-2

Introduction to the Embedded Target for Infineon C166® Microcontrollers 1-3
Feature Summary 1-3
Configuring for Different Hardware Variants and Memory Models 1-4

Hardware and Software Requirements 1-5
Host Platform 1-5
Hardware Requirements 1-5

Software Requirements **1-5**

Setting Up and Verifying Your Installation **1-6**

Setting Up Your Target Hardware **1-7**
Jumper Settings for the phyCore-167 Development Board . . . **1-7**

Setting Target Preferences **1-8**

Customizing the Build Process **1-10**

Supported Blocks and Data Types **1-12**

2 Tutorial: Simple Example Applications for C166® Microcontrollers

Introduction **2-2**

Tutorial: Creating a New Application **2-3**
Before You Begin **2-3**
Example Model 1: c166_serial_transmit **2-3**
Generating and Downloading Code **2-6**
Verifying Code Execution on the Target **2-9**
Troubleshooting: MiniMon Settings **2-9**
Example 2: c166_serial_io **2-11**

Starting the Debugger
on Completion of the Build Process **2-13**
Fixed-Point Example Model: c166_fuelsys **2-15**

Generating ASAP2 Files **2-17**

3 Integrating Your Own Device Drivers

Integrating Hand-Coded Device Drivers with a Simulink Model . . . 3-2

Preparing Input and Output Signals to the Device Driver Functions . . . 3-3

Calling the Device Driver Functions from c166_main.c . . . 3-6

Adding the I/O Driver Source to the List of Files to Build . 3-8

Tutorial: Using the Example Driver Functions . . . 3-9

4 Custom Storage Class for C166® Microcontroller Bit-Addressable Memory

Specifying C166® Microcontroller Bit-Addressable Memory 4-2

Using the Bitfield Example Model . . . 4-3

5 Block Reference

The Embedded Target for Infineon C166® Microcontrollers Block Library . . . 5-2

Using Block Reference Pages . . . 5-2

Blocks Organized by Library . . . 5-3

C166 Drivers Library . . . 5-3

Data Type Support and Scaling for

Device Driver Blocks **5-4**
Configuration Class Blocks **5-5**

Alphabetical List of Blocks **5-7**

Preface

This section includes the following topics:

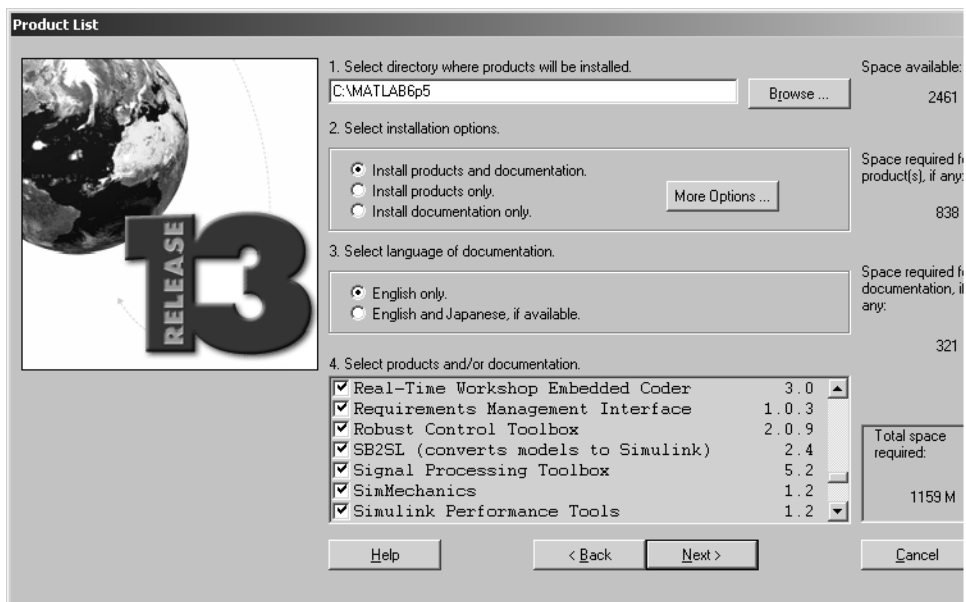
Installing the Embedded Target for Infineon C166® Microcontrollers (p. vi)	Installation of the product.
Using This Guide (p. vii)	Suggested path through this document to get you up and running quickly with the Embedded Target for Infineon C166® Microcontrollers.
Required Products (p. viii)	Products required when using the Embedded Target for Infineon C166® Microcontrollers; also products that are especially relevant to the kinds of tasks you can perform with the Embedded Target for Infineon C166® Microcontrollers.
Typographical Conventions (p. x)	Formatting conventions used in this document.

Installing the Embedded Target for Infineon C166® Microcontrollers

Your platform-specific MATLAB Installation Guide provides all of the information you need to install the Embedded Target for Infineon C166® Microcontrollers.

Prior to installing the Embedded Target for Infineon C166® Microcontrollers, you must obtain a License File or Personal License Password from The MathWorks. The License File or Personal License Password identifies the products you are permitted to install and use.

As the installation process proceeds, it displays a dialog similar to the one below, letting you indicate which products to install.



Using This Guide

We suggest the following path to get acquainted with the Embedded Target for Infineon C166® Microcontrollers and gain hands-on experience with the features most relevant to your interests:

- Read Chapter 1, “Product Overview” in its entirety, paying particular attention to “Setting Up and Verifying Your Installation” on page 1-6.
- If you are interested in using the device driver blocks supplied with Embedded Target for Infineon C166® Microcontrollers and in deploying stand-alone, real-time applications on the C166®, read Chapter 2, “Tutorial: Simple Example Applications for C166® Microcontrollers.” Work through the “Tutorial: Creating a New Application” on page 2-3.
- Then, if you are interested in using Embedded Target for Infineon C166® Microcontrollers for integrating automatically generated code with your own hand-written device driver code, see “Integrating Hand-Coded Device Drivers with a Simulink Model” on page 3–2. Work though the example provided in “Tutorial: Using the Example Driver Functions” on page 3–9.
- See “Custom Storage Class for C166® Microcontroller Bit-Addressable Memory” on page 4–1 to find out how to use Embedded Target for Infineon C166® Microcontrollers to take advantage of C166® bit-addressable memory. This can significantly reduce code size and increase execution speed. There are examples provided in “Using the Bitfield Example Model” on page 4–3.
- For in-depth information about the device drivers and other blocks supplied with Embedded Target for Infineon C166® Microcontrollers, see Chapter 5, “Block Reference.” It is particularly important to read “C166 Resource Configuration” on page 5-9, as the C166 Resource Configuration block is required to use the device driver blocks.
- See also “Embedded Target for Infineon C166® Microcontrollers Demos” on page -xii.

Required Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the Embedded Target for Infineon C166® Microcontrollers. They are listed in the table below.

The Embedded Target for Infineon C166® Microcontrollers *requires* these products:

- MATLAB® 6.5 (Release 13)
- Simulink® 5.0 (Release 13)
- Real-Time Workshop® 5.0 (Release 13)
- Real-Time Workshop Embedded Coder 3.0 (Release 13)

For more information about any of these products, see either

- The online documentation for that product, if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section

Related Products

Note The toolboxes listed below all include functions that extend the capabilities of MATLAB. The blocksets include blocks that extend the capabilities of Simulink.

Product	Description
Fixed-Point Blockset	Design and simulate fixed-point systems
MATLAB	The Language of Technical Computing
Real-Time Workshop	Generate C code from Simulink models

Product	Description
Real-Time Workshop Embedded Coder	Generate production code for embedded systems
Simulink	Design and simulate continuous- and discrete-time systems
Stateflow®	Design and simulate event-driven systems
Stateflow Coder	Generate C code from Stateflow charts

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, and user input	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'`**`whole`**`')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace font`	MATLAB responds with
		`A =`
		`5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'`*`method`*`')`

Demos

This section includes the following topics:

Embedded Target for Infineon C166® Microcontrollers Demos (p. xii)	Hyperlinks to demo models that illustrate product features and how to use them.

Embedded Target for Infineon C166® Microcontrollers Demos

We have provided some demos to help you become familiar with features of the Embedded Target for Infineon C166® Microctrontrollers.

If you are reading this document online in the MATLAB® Help browser, you can start the demos by clicking on the links in the **Command** column of the following table.

Alternatively, you can access the demo suite by typing commands from the **Command** column of the table, at the MATLAB command prompt, as in this example:

```
c166_serial_transmit
```

Embedded Target for Infineon C166® Microcontrollers Demos

Command	Demo Topic
c166_serial_transmit	Simple model demonstrating use of serial transmit, a device driver block provided with the Embedded Target for Infineon C166® Microcontrollers real-time target. See "Tutorial: Creating a New Application" on page 2-3.
c166_serial_io	Simple model demonstrating use of serial transmit and receive blocks. See "Tutorial: Creating a New Application" on page 2-3

Embedded Target for Infineon C166® Microcontrollers Demos (Continued)

Command	Demo Topic
`c166_user_io.mdl`	This example model demonstrates how to integrate user-defined device driver code. You can generate code from the `controller` subsystem, which will automatically download and run on the target. If you are using a Phytec phyCORE module with HD200 development board, you can see successful execution of the code when the LED blinks. Prior to generating code, you can run this model in closed-loop simulation. If you use this model as a basis for integrating your own device driver code, this closed-loop simulation allows you to validate the correct behavior of your control algorithm before running it in real-time. Follow the links in the model to open the source files for the device driver code; you can use these as a guide for writing your own device drivers. See "Tutorial: Using the Example Driver Functions" on page 3-9. This demo is also used as example to show you how to start your debugger as part of the build process. See "Starting the Debugger on Completion of the Build Process" on page 2–13.

Embedded Target for Infineon C166® Microcontrollers Demos (Continued)

Command	Demo Topic
`c166_fuelsys.mdl`	This fixed-point demo model was derived from the floating-point demo `fuelsys.mdl` (for Stateflow). The floating-point control algorithm from this original model has been converted to fixed point in order to allow efficient code generation for the C166® microcontroller. This demo starts the debugger at the end of the build in simulation mode rather than on-chip. The modifications reducing RAM and ROM requirements are described in “Fixed-Point Example Model: c166_fuelsys” on page 2-15. Note this demo requires Stateflow®, Stateflow Coder and the the Fixed-Point blockset.
`c166_bitfields.mdl`	This model uses a custom storage class (provided with Embedded Target for Infineon C166® Microcontrollers) to take advantage of C166® microcontroller bit-addressable memory. This can significantly reduce code size and increase execution speed. This model is also configured to start the debugger at the end of the build. See “Using the Bitfield Example Model” on page 4–3, which includes a comparison with another custom storage class variable in the `c166_user_io` model. See “Custom Storage Class for C166® Microcontroller Bit-Addressable Memory” on page 4–1.

1

Product Overview

This section contains the following topics:

Prerequisites (p. 1-2)	What you need to know before using the Embedded Target for Infineon C166® Microcontrollers.
Introduction to the Embedded Target for Infineon C166® Microcontrollers (p. 1-3)	Overview of the product and the use of the Embedded Target for Infineon C166® Microcontrollers in the development process.
Hardware and Software Requirements (p. 1-5)	Hardware platforms supported by the product; development tools (e.g. compilers, debuggers) required for use with the product.
Setting Up and Verifying Your Installation (p. 1-6)	Overview of setting up your development tools and hardware to work with the Embedded Target for Infineon C166® Microcontrollers, and verifying correct operation.
Setting Up Your Target Hardware (p. 1-7)	Port connections and jumper settings.
Setting Target Preferences (p. 1-8)	Configuring environmental settings and preferences associated with the Embedded Target for Infineon C166® Microcontrollers.
Customizing the Build Process (p. 1-10)	This section explains the purpose of each of the configuration files specified in the C166® Target Preferences. You will need to understand these if you want to change the default settings supplied with the Embedded Target for Infineon C166® Microcontrollers.
Supported Blocks and Data Types (p. 1-12)	Requirements and restrictions

Prerequisites

This document assumes you are experienced with MATLAB®, Simulink®, Real-Time Workshop®, and the Real-Time Workshop Embedded Coder.

Minimally, you should read the following from the “Basic Concepts and Tutorials” section of the Real-Time Workshop documentation:

- “Basic Real-Time Workshop Concepts.” This section introduces general concepts and terminology related to Real Time Workshop.
- “Quick Start Tutorials.” This section provides several hands-on exercises that demonstrate the Real-Time Workshop user interface, code generation and build process, and other essential features.

You should also familiarize yourself with the Real-Time Workshop Embedded Coder documentation.

In addition, if you want to understand and use the device driver blocks in the the Embedded Target for Infineon C166® Microcontrollers library, you should have at least a basic understanding of the architecture of the C166®. The C166 *Users Manual* (or corresponding document for your C166® derivative processor) is required reading. We recommend that you read the introduction to the C166® microcontroller. You can find this document by searching the Infineon web site for the C166® family of microcontrollers, at the following URL:

`http://www.infineon.com/`

Introduction to the Embedded Target for Infineon C166® Microcontrollers

The Embedded Target for Infineon C166® Microcontrollers is an add-on product for use with the Real-Time Workshop Embedded Coder. It provides a set of tools for developing embedded applications for the C166® family of processors. This includes derivatives such as Infineon C167 and ST Microelectronics ST10 (`www.us.st.com`).

Used in conjunction with Simulink, Stateflow, and the Real-Time Workshop Embedded Coder, the Embedded Target for Infineon C166® Microcontrollers lets you

- Design and model your system and algorithms.
- Compile, download, run and debug generated code on the target hardware, seamlessly integrating with industry-standard compilers and development tools for the C166® microcontroller.
- Use rapid prototyping techniques to evaluate performance and validate results obtained from generated code running on the target hardware.
- Deploy production code on the target hardware.

Feature Summary

- Automatic generation of ‘`main`’ program including single or multitasking scheduler
- Automated build procedure including starting debugger or download utility
- Support for integer, floating-point or fixed-point code
- Driver blocks for serial transmit and receive
- Examples to show you how to integrate your own driver code
- Fully integrated with Tasking toolchain
- Enhanced HTML report generation provides analysis of RAM/ROM usage; this is in addition to the standard HTML report generation that shows optimization settings and hyperlinks to generated code files

Configuring for Different Hardware Variants and Memory Models

There are many different members of the C166® microcontroller family, e.g. C167CR, C167CS, ST10. For each of these processors, it is appropriate to use different compiler switches and link libraries.

The Embedded Target for Infineon C166® Microcontrollers is supplied with default configurations that have been tested on the following hardware:

- Phytec phyCORE-167 ST10 26 9
- Phytec phyCORE-167 C167S
- Phytec kitCON-167 C167R

If your hardware variant is not on this list you may need to change the default configuration settings. To do this you should consult your hardware and compiler documentation. These settings can then be applied to Embedded Target for Infineon C166® Microcontrollers by creating new versions of the following files:

- Tasking Make Variables are specified in these files:
  - `C167-S.mk` — includes details of which libraries to link against, which memory model, and which compiler switches
  - `c166_rtwec.ilo` — instructions to C166 locator program on where to place program code in memory; this file is specified in `C167-S.mk`.
- Tasking Startup Macros are for configuration of startup code.
  - Use `kc167.inc` for kitCON-167
  - Use `ph167.inc` for phyCORE-167 boards.

You may want to use the flags from the Tasking auto-generated makefile.

There is a possibility of setting up an Embedded Development Environment (EDE) project, then setting **Generate code only** in **RTW Options**. The build process can then be controlled from EDE. This term is specific to the Tasking toolset.

Hardware and Software Requirements

Host Platform

The Embedded Target for Infineon C166® Microcontrollers supports only the PC platform: Windows 2000, NT and XP only.

Hardware Requirements

Embedded Target for Infineon C166® Microcontrollers may be used to generate programs that can run on any development board or Electronic Control Unit (ECU) that is based on the C166® microcontroller.

In this document, however, we assume that you are working with the Phytec phyCORE-167CS development board, and we document specific settings and procedures for use with the Phytec phyCORE-167CS board, in conjunction with specific cross-development environments.

If you use a different development board, you may need to adapt these settings and procedures for your development board.

Software Requirements

See “Required Products” for information on MathWorks products required to use Embedded Target for Infineon C166® Microcontrollers.

In addition to the required MathWorks software, a supported cross-development environment is required. The Embedded Target for Infineon C166® Microcontrollers currently supports the cross-development tools listed below:

- Tasking CrossView compiler and debugger toolchain
- MiniMon freeware download and monitor utility

Before using the Embedded Target for Infineon C166® Microcontrollers with the above cross-development tools, please be sure to read and follow the instructions in “Setting Up and Verifying Your Installation” on page 1-6.

Setting Up and Verifying Your Installation

The next sections describe how to configure your development environment (compiler, debugger, etc.) for use with the Embedded Target for Infineon C166® Microcontrollers and verify correct operation. The initial configuration steps are described in the following sections:

- "Setting Up Your Target Hardware" on page 1–7
- "Setting Target Preferences" on page 1–8

Install Tasking compiler and CrossView debugger by following the instructions provided by Altium Limited.

You can obtain the MiniMon download utility for monitoring the serial interface at this URL:

`http://www.infineon.com`

Be sure to install the 2.1.19 version. Earlier versions do not contain all the required controller configurations.

Setting Up Your Target Hardware

In this document, we assume that you are working with the phyCORE-167CS module with HD200 development board. This section describes the required connections and jumper settings for the board.

After setting up your board, you must configure target settings associated with the Embedded Target for Infineon C166® Microcontrollers, as described in the next section.

Connect the supplied power cable to the board, and use the serial cable to connect the serial port P1 on the board to the serial port of your PC.

Jumper Settings for the phyCore-167 Development Board

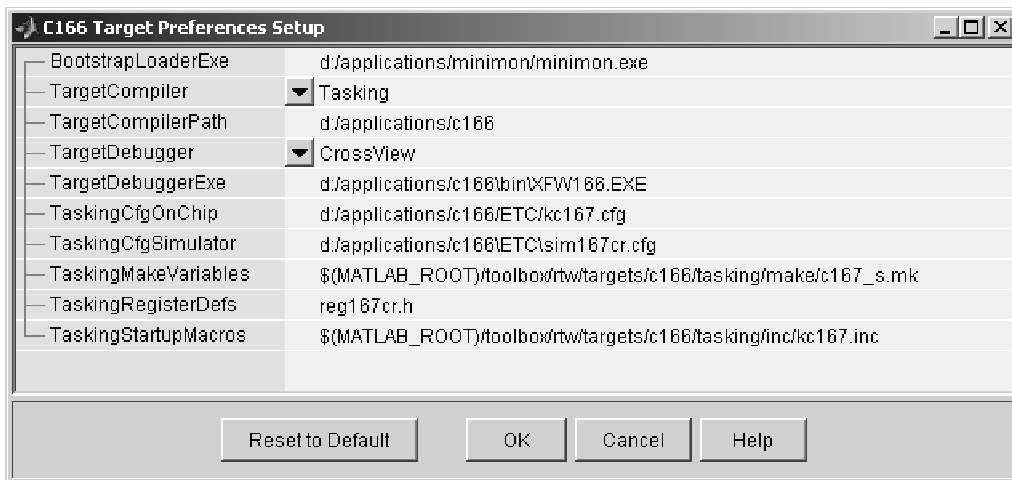
1 Configure jumpers as detailed in the instructions found in the phyCORE QuickStart documentation. Note that we have found these settings to be markedly different to the configuration fresh out of the box.

2 It is useful if the board starts up in bootloader rather than execution mode. There is one jumper setting that needs to be changed to achieve this: close pins 1 and 2 on JP10. This is optional; if you do not close this jumper, then when you download to the target you need to keep the Boot switch depressed while pressing the Reset button.

Setting Target Preferences

This section describes configuration settings associated with the Embedded Target for Infineon C166® Microcontrollers. These settings, which persist across MATLAB sessions and different models, are referred to as *target preferences*. Target preferences let you specify the location of your cross-compiler and other parameters affecting the generation, building, and downloading of code:

- Start the Target Preferences Setup GUI by selecting **Start –> Simulink –> Embedded Target for Infineon C166® Microcontrollers –> C166 Target Preferences**.



Here you can edit the settings for your cross-development environment:

- `BootstrapLoaderExe` specifies the path to your download utility (MiniMon).
- `TargetCompiler` and `TargetCompilerPath` specify the name and path to your compiler (Tasking)
- `TargetDebugger` and `TargetDebuggerExe` specify the name and path to your debugger executable (CrossView)
- `TaskingCfgOnChip` specifies the name of a CrossView configuration file that will be used to start CrossView when the build action is set to

`Download_and_run_with_debugger` (see “CrossView Configuration Files” on page 1–10).

- `TaskingCfgSimulator` specifies the name of a CrossView configuration file that will be used to start CrossView when the build action is set to "Run_with_simulator" (see “CrossView Configuration Files” on page 1–10).
- `TaskingMakeVariables` specifies a file that contains compiler flags, linker flags and other build settings (see “Tasking Make Variables” on page 1–10).
- `TaskingRegisterDefs` specifies an include file that may be used in the automatically generated C source code; this file should be located in the `include` sub-directory of your Tasking compiler installation.
- `TaskingStartupMacros` specifies a file containing macro definitions for the startup code used by the Tasking compiler (see “Configuring the Tasking Startup Code” on page 1–11).

You must check these paths are correct for your machine. You may need to localize these paths to suit your PC. You can edit a path by clicking on it. The drive designated in the path must be either an actual hard drive on your PC, or a mapped drive. Do not use a Universal Naming Convention (UNC).

See the next section, “Customizing the Build Process” on page 1–10 for more information on using the target preferences.

Customizing the Build Process

The default settings provided with Embedded Target for Infineon C166® Microcontrollers allow you to build an application using the C166® microcontroller small memory model and with registers configured appropriately for a number of evaluation boards, including the Phytec phyCORE-C167CS, and the Phytec KC167.

You can change the default settings by supplying your own configuration files in the **C166 Target Preferences Setup** dialog. The following information explains the purpose of each of the configuration files.

CrossView Configuration Files

The CrossView configuration files are specified by the Target Preferences `TaskingCfgOnChip` and `TaskingCfgSimulator`. The CrossView configuration files will typically be selected from the ones supplied with the CrossView debugger. Two separate configuration files may be specified; these are only used if one of the options `Run_with_simulator` or `Download_and_run_with_debugger` is specified in the C166® microcontroller section of the RTW options. Consult the CrossView documentation "C166/ST10 v7.5 CROSSVIEW PRO DEBUGGER USER'S GUIDE" for further details.

Tasking Make Variables

The target preference `TaskingMakeVariables` allows a make file to be specified that is included in the main make file that is run during the build process. You can inspect the default make settings file `matlabroot/toolbox/rtw/targets/c166/tasking/make/c167_s.mk` for an example of the make variables that must be specified. Consult the Tasking User Guides "C166/ST10 v7.5 C CROSS–COMPILER USER'S GUIDE" and "C166/ST10 v7.5 CROSS–ASSEMBLER, LINKER/LOCATOR, UTILITIES USER'S GUIDE" for further details.

Tasking Register Definitions File

The target preference `TaskingRegisterDefs` allows you to select a register definitions file that is appropriate for your target hardware. This file is normally located in the include directory under the Tasking installation directory; it contains definitions of all the special function registers etc., that may be dependent upon your target hardware. See the Tasking User Guide "C166/ST10 v7.5 C CROSS–COMPILER USER'S GUIDE" for further details.

Configuring the Tasking Startup Code

The Tasking toolchain allows the startup code to be configured according to the hardware you are using. This is achieved by defining macros that are interpreted by the macro preprocessor `m166`. By defining appropriate macros, you can set hardware-dependent initial values for certain C166® registers. Note that when you are running your program using the CrossView debugger, these startup settings will generally be overridden by CrossView using values for the CrossView configuration files described above.

For more details on configuring the Startup Code, consult "C166/ST10 v7.5 C CROSS–COMPILER USER'S GUIDE" or try using the Infineon Digital Application Engineer DAvE.

Supported Blocks and Data Types

Embedded Target for Infineon C166® Microcontrollers supports the same blocks and data types as Real-Time Workshop Embedded Coder.

Note however

1. You should not use IEEE values `Inf` or `NaN` in your model: these are not supported and will result in an error.

2. Floating point support is implemented in the software; if speed and ROM usage are of concern you should select the option for integer code and avoid the use of floating-point values in your model. This is detailed in step 9 of "Tutorial: Using the Example Driver Functions" on page 3-9.

2

Tutorial: Simple Example Applications for C166® Microcontrollers

This section includes the following topics:

Introduction (p. 2-2)	An overview of the Embedded Target for Infineon C166® Microcontrollers real-time target, other components required to generate stand-alone real-time applications, and the process of deploying generated code on target hardware.
Tutorial: Creating a New Application (p. 2-3)	A hands-on exercise in building two simple applications from demo models, including downloading and executing generated code on a target board.
Starting the Debugger on Completion of the Build Process (p. 2-13)	This exercise shows you how to generate code and commence debugging automatically as part of the build process. Depending on your debugger, you can debug the application either on-chip or on a hardware simulator.
Generating ASAP2 Files (p. 2-17)	How to generate ASAP2 files for your models.

Introduction

This section describes how to use two example models to generate, download and run stand-alone real-time applications for the C166® microcontroller. The components required to generate stand-alone code are

- The Embedded Target for Infineon C166® Microcontrollers real-time target
- The example models provided: `c166_serial_transmit.mdl` and `c166_serial_io.mdl`
- The Tasking Cross View compiler and debugger and MiniMon download utility for compiling and downloading generated code to the target hardware

Using these, you can build the complete applications. You do not need to hand-write any C code to integrate the generated code into a final application.

The tutorial "Tutorial: Creating a New Application" on page 2-3 uses two blocks from the Embedded Target for Infineon C166® Microcontrollers library. For complete information on the Embedded Target for Infineon C166® Microcontrollers library blocks, see "Block Reference" on page 5–1.

Tutorial: Creating a New Application

In this tutorial, you will build stand-alone real-time applications from models incorporating blocks from the Embedded Target for Infineon C166® Microcontrollers library. We assume that you are already familiar with Simulink and with the Real-Time Workshop code generation and build process.

In the following sections, you will

- Examine two models
- Generate code from the models
- Download and run the code automatically as part of the build process
- Use MiniMon to monitor the code executing on the target
- Use the Cross View debugger to run a model on the C166 Simulator or debug on-chip

Before You Begin

This tutorial requires the following specific hardware and software in addition to the Embedded Target for Infineon C166® Microcontrollers:

- Phytec phyCORE-167CS development board, connected via serial port to your PC
- Tasking Cross View compiler and debugger
- MiniMon download utility

You must make sure the target preferences have been set correctly. See “Setting Target Preferences” on page 1-8.

Note Make sure the `default.ini` file in the MiniMon directory is not read only. This can cause errors.

Example Model 1: c166_serial_transmit

In this tutorial you will start with a simple example model, `c166_serial_transmit`, from the directory *`matlabroot`*`/toolbox/rtw/targets/c166/c166demos`.

This directory is on the default MATLAB path:

1 Open the model by typing `c166_serial_transmit` at the command line.

This example shows the tutorial model `c166_serial_transmit` at the root level.

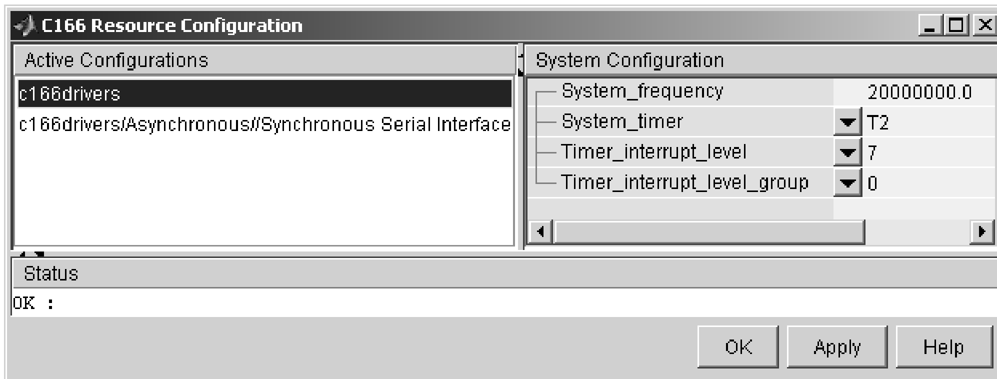
The model contains a C166 Resource Configuration object. When building a model with driver blocks from the Embedded Target for Infineon C166® Microcontrollers library, you must always place a C166 Resource Configuration object into the model (or the subsystem from which you want to generate code) first.

The purpose of the C166 Resource Configuration object is to provide information to other blocks in the model. Unlike conventional blocks, the C166 Resource Configuration object is not connected to other blocks via input or output ports. Instead, driver blocks (such as the ASCO Serial Transmit block in the example model) query the C166 Resource Configuration object for required information.

For example, a driver block may need to find the system clock speed that is configured in the C166 Resource Configuration object. The C166® microcontroller has a number of clocked subsystems; to generate correct code, driver blocks need to know the speeds at which the these clock busses will run.

The C166 Resource Configuration window lets you examine and edit the C166 Resource Configuration settings.

2 To open the C166 Resource Configuration window, double-click on the C166 Resource Configuration icon. The picture following shows the C166 Resource Configuration window for the `c166_serial_transmit` model.

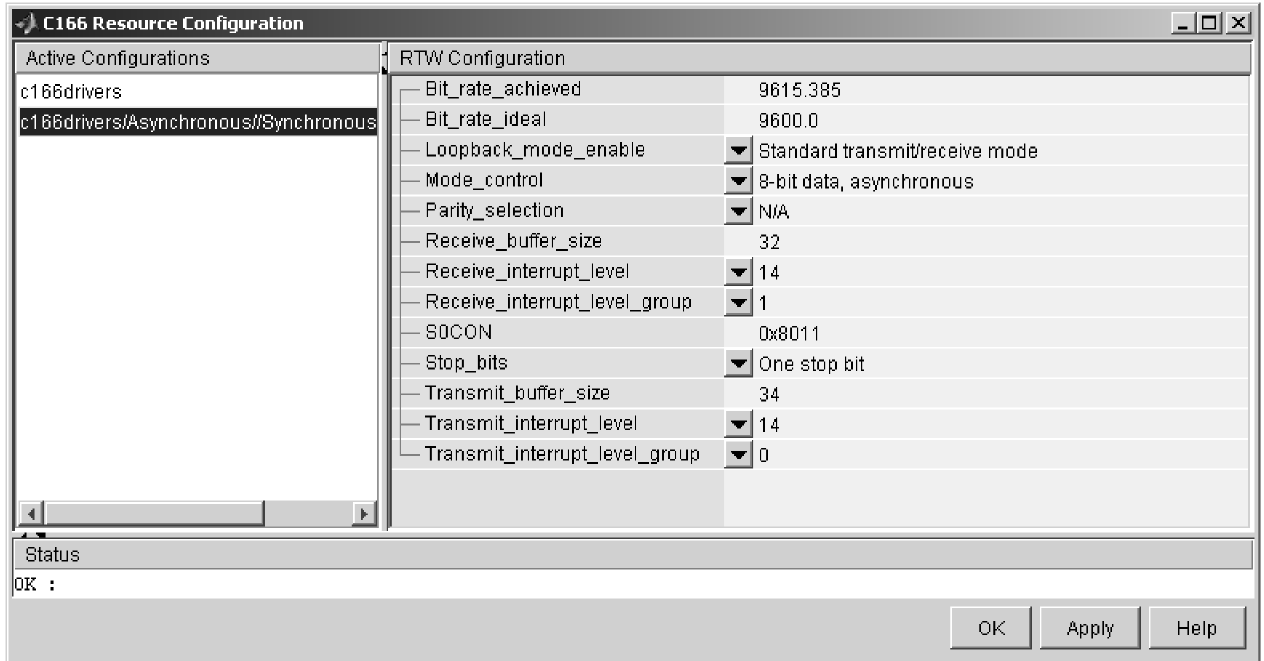


In this tutorial, you will use the default C166 Resource Configuration settings.

Note If hardware is running at a system frequency other than 20MHz you must change this parameter appropriately.

Otherwise, observe, but do not change, the parameters in the **C166 Resource Configuration** window. By default the `c166drivers` configuration is selected. This shows parameters for the C166® microcontroller CPU in the **System Configuration** pane on the right.

You can see the settings for the serial driver block by clicking on the `c166drivers/Asynchronous/Synchronous Serial Interface` option in the **Active Configurations** pane. These settings are shown in the following illustration.



The settings appear in the **RTW Configuration** pane on the right. Do not edit any of these parameters for this tutorial. To learn more about the C166 Resource Configuration object, see "C166 Resource Configuration" on page 5-9.

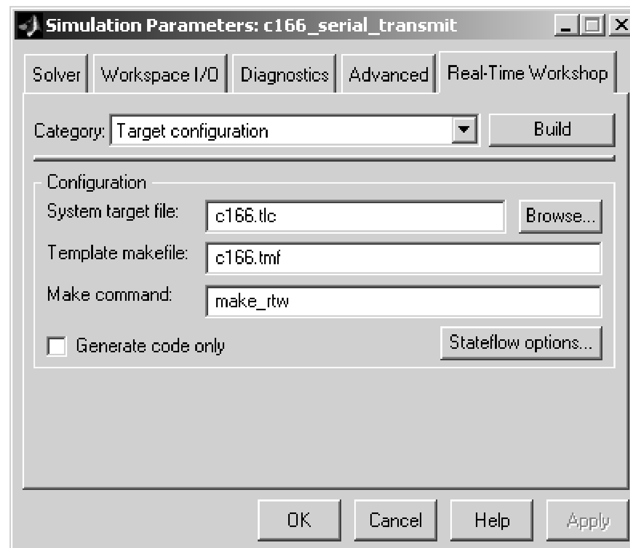
Close the **C166 Resource Configuration** window before proceeding.

Generating and Downloading Code

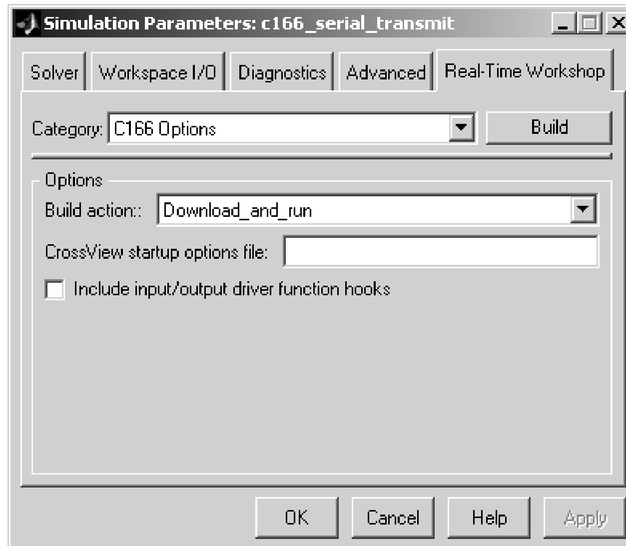
To generate code for the model:

1 Select **Simulation –> Simulation Parameters**.

The **Simulation Parameters** dialog opens.



2 From the **Category** drop-down menu, select `C166 Options`.



Observe that the **Build action** is `Download_and_run`. When you generate code for this model, it will automatically start a download utility program and load the application onto C166® microcontroller hardware over a serial connection. The code will then begin execution on the target.

3 Click **Build**.

Note that you could have gone straight to building the model by selecting **Tools** –> **Real-Time Workshop** –> **Build Model** or using the short cut **Ctrl+B**.

Watch the progress messages in the command window as code is generated. When MiniMon is started, a dialog appears asking you to reset your hardware.

4 Press the Reset button on your phyCORE-167CS board or cycle the power, and then click **OK**.

You can see progress messages in the MiniMon window as it connects and then downloads to the target. MiniMon will then disappear and the code will begin executing on the target.

Verifying Code Execution on the Target

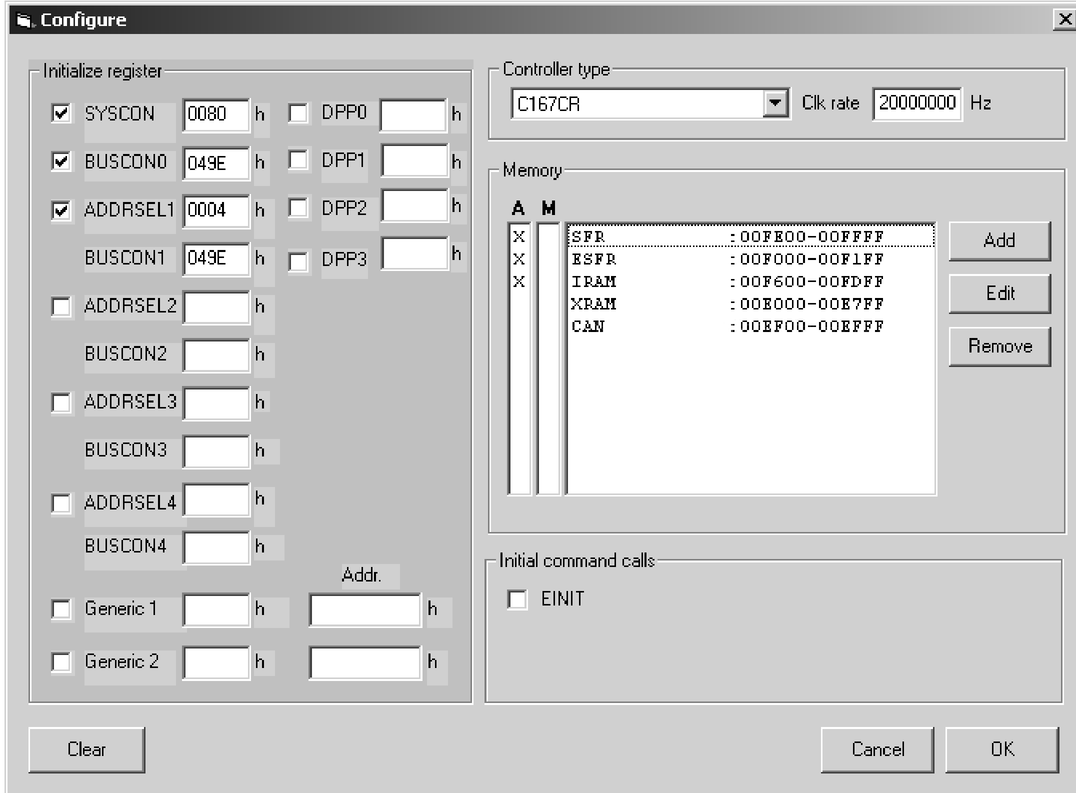
1. Start MiniMon (select **Start –> Programs –> MiniMon –> MiniMon** in Windows, or navigate to `MiniMon.exe` and double-click).
2. Watch the model output in the MiniMon window. When the application is running it sends the text “Hello World” plus a carriage return plus a linefeed over the serial interface.

Troubleshooting: MiniMon Settings

If you have a problem downloading, click Configure Hardware ( )in the toolbar (or select **Target –> Configuration...**) and check that MiniMon has the correct target settings, as illustrated below.

This configuration has been verified with both a phyCORE C167CS board and a Phytec kc167 (C167CR).

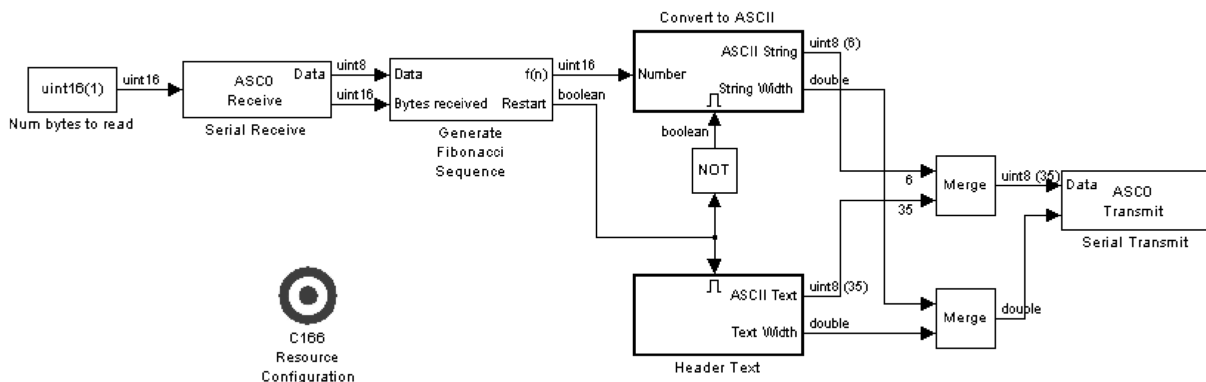
It may be necessary to change the **Controller type** depending on your hardware. For example, if you are using a phyCORE-167 with ST10F269 processor you should select **Controller type** `C164CH`. This works because both these controllers share the same bootstrap loader identification byte.


Configure
Initialize register
SYSCON 0080 h
BUSCON0 049E h
ADDRSEL1 0004 h
BUSCON1 049E h
ADDRSEL2 h
BUSCON2 h
ADDRSEL3 h
BUSCON3 h
ADDRSEL4 h
BUSCON4 h
DPP0 h
DPP1 h
DPP2 h
DPP3 h
Addr.
Generic 1 h h
Generic 2 h h
Controller type
C167CR
Clk rate 20000000 Hz
Memory
A M
X SFR :00FE00-00FFFF
X ESFR :00F000-00F1FF
X IRAM :00F600-00FDFF
XRAM :00E000-00E7FF
CAN :00EF00-00EFFF
Add
Edit
Remove
Initial command calls
EINIT
Clear
Cancel
OK

Example 2: c166_serial_io

We provide another example model which demonstrates how to use both serial transmit and receive blocks for the C166® microcontroller. You could use these blocks in this way with your own Simulink models.

1 Open the model by typing `c166_serial_io` at the command line.



2 Press **Ctrl+B** or select **Tools –> Real-Time Workshop –> Build Model**.

Watch the progress messages as code is generated from the model and MiniMon is automatically started to download the code to the target over the serial connection. The MiniMon dialog appears asking you to reset your hardware.

3 Press the Reset button on your phyCORE-167CS board or cycle the power, and then click **OK**.

You can see progress messages in the MiniMon window as it connects and then downloads to the target. MiniMon will then disappear and the code will begin executing on the target.

You can restart MiniMon to monitor the serial interface.

Verifying Code Execution on the Target

1 Start MiniMon (select **Start ->Programs ->MiniMon -> MiniMon** in Windows, or navigate to `MiniMon.exe` and double-click).

2 Watch the model output in the MiniMon window. When the application is running, it generates a sequence of 16 bit numbers, converts them to ASCII characters, and transmits them over the serial interface.

3 If you enter the character `'r'` in the MiniMon command line field, the application will restart at the beginning of the sequence. Examine the model to see how this works: the serial receive block passes the restart command though to the `Generate Fibonacci Sequence` subsystem. This subsystem checks for the restart command.

Starting the Debugger on Completion of the Build Process

As an alternative to downloading with MiniMon at the end of the build process, you can start your debugger. Depending on the features provided by your debugger, you can debug the application either on-chip or on a hardware simulator.

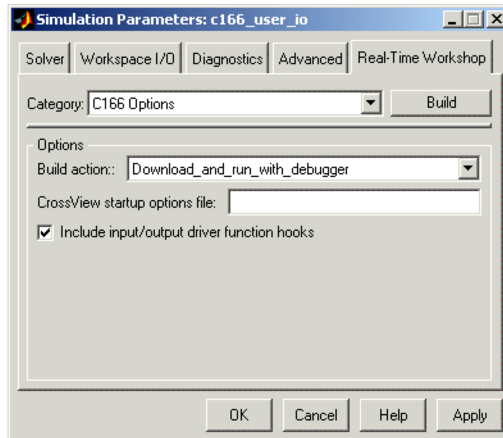
For this example you will use another demo model, `c166_user_io.mdl`. This model is designed to show you how to integrate your own hand-coded device drivers with automatically generated code using Embedded Target for Infineon C166® Microcontrollers. This model is covered in detail in Chapter 3, “Integrating Your Own Device Drivers.” You will use it as an example here because you will typically need to use the debugger in cases where you are integrating your own code.

Also, note that running the debugger on-chip over the serial interface will conflict with the serial transmit and receive blocks. The `c166_user_io` model does not use serial blocks, so this avoids serial conflicts for this example. If you need to debug an application that includes the serial transmit and receive blocks, you must run the debugger using a hardware simulator; alternatively, it may be possible to run your debugger on-chip without using the serial interface, for example if debugging over CAN is available.

1 Open the model `c166_user_io.mdl`.

2 Select **Simulation** –> **Simulation parameters**.

3 Select `C166 Options` from the **Category** drop-down menu.



4 Select the build action `Run_with_simulator` or `Download_and_run_with_debugger`.

5 Before generating code, check that your target preferences related to the debugger are correctly configured. See “Setting Target Preferences” on page 1–8.

6 Click **OK**.

7 Right click on the `controller` subsystem and select **Real-Time Workshop –> Build Subsystem**.

8 Click **Build** in the following dialog.

   Watch the progress messages in the command window as code is generated. At the end of the build process your debugger will be launched automatically with the application ready to run. You may now debug the application.

Note If your model contains a serial transmit or receive block, it is not possible to perform on-chip debugging over the same serial interface. Attempting to use the debugger in this case will cause an error.

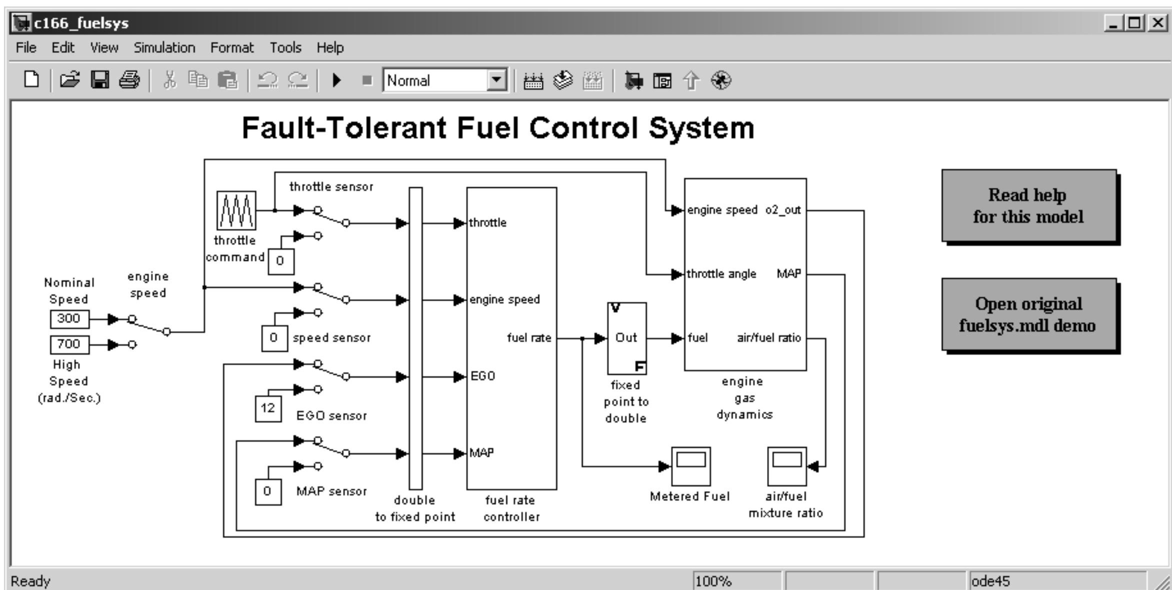
Fixed-Point Example Model: c166_fuelsys

This demo model was derived from the demo `fuelsys.mdl`. The floating point control algorithm from this original model has been converted to fixed point in order to allow efficient code generation for the Infineon C166® microcontroller. This demo starts the debugger in simulation mode rather than on-chip.

Note this demo requires Stateflow®, Stateflow Coder and the Fixed-Point Blockset.

The complete model includes includes a plant simulation as well as a fixed point implementation of the control algorithm. When you generate code for this example, be sure to generate code for the control algorithm sub-system only:

1 Open the model `c166_fuelsys.mdl`.



2 Right-click on the block `fuel rate controller`

3 From the pop-up menu, select **Real Time Workshop –> Build Subsystem**.

4 On the following dialog click **Build**.

When code generation is complete, the Code Generation Report appears in your Help Browser. Here you can review the RAM and ROM requirements of the model. To do this, left-click on the link `Code profile report` in the left list. For comparison, you may want to build the original floating point version of the fuelsys control algorithm: you should find that using the fixed point implementation results in a considerable reduction in both RAM and ROM.

Note that in the fixed point version of the `fuelsys` model, RAM and ROM requirements have been reduced by:

- Selecting in-line parameters (on the **Advanced** tab of the **Simulation Parameters** dialog)
- Using C166 bit-addressable memory for some signals with datatype boolean (see “Custom Storage Class for C166® Microcontroller Bit-Addressable Memory” on page 4–1 for a detailed example)
- Replacing the reference to time, `t`, inside the Stateflow chart with a counter (this is necessary in order to create integer only code)
- Skipping the index search. The variable `press` is used in several lookup tables; because the values in this vector are evenly spaced the generated code is optimized by skipping the index search; to ensure that the conversion to fixed point does not affect this optimization, the variable `press` must be replaced by `c166_fixpt_evenspace_cleanup(press,sfix(16),2^-14)`; this function makes an adjustment to the input values to ensure that they will still be evenly spaced after conversion to fixed point.
- Switching off the option **Saturate on integer overflow** for all Sum, Product, Switch and Look-Up Table blocks inside the `fuel rate controller` subsystem. This is a check-box option (you must click **Show additional parameters** to see it).

Further reductions in RAM and ROM are possible by changing the lookup-method in some or all of the look-up tables in this model. For example, by selecting `Use input below` instead of `Interpolation-Use End Values` a significant reduction in memory requirement is possible; this further optimization should only be considered if the degradation in performance is deemed to be acceptable.

The example model `c166_bitfields.mdl` is also configured to launch the debugger at the end of the build. See “Using the Bitfield Example Model” on page 4-3 for details.

Generating ASAP2 Files

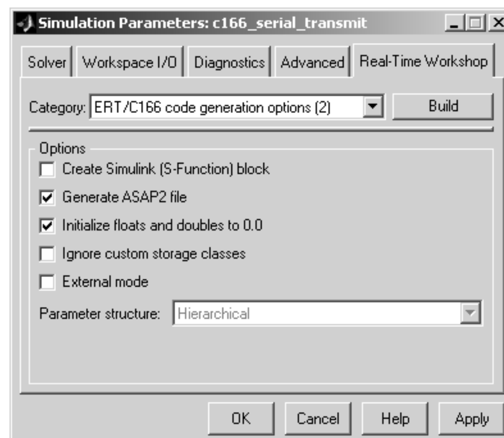
ASAP2 is a data definition standard proposed by the Association for Standardization of Automation and Measuring Systems (ASAM). ASAP2 is a standard description for data measurement, calibration, and diagnostic systems. The Embedded Target for Infineon C166® Microcontrollers lets you export an ASAP2 file containing information about your model during the code generation process.

Before you begin generating ASAP2 files with the Embedded Target for Infineon C166® Microcontrollers, you should read the "Generating ASAP2 Files" section of the Real-Time Workshop Embedded Coder documentation. That section describes how to define the signal and parameter information required by the ASAP2 file generation process.

Select the ASAP2 option before the build process as follows:

1 Select **Simulation** –> **Simulation Parameters**.

The **Simulation Parameters** dialog appears.



2 On the Real Time Workshop tab, select `ERT/C166 code generation options (2)` from the **Category** drop-down menu.

3 Select the **Generate ASAP2 file** option.

4 Click **Apply**.

As part of the build process, an ASAM compliant ASAP2 data definition file will be created for the generated C code.

Note that standard Real-Time Workshop ASAP2 file generation does not include the memory address attributes in the generated file. Instead it leaves a placeholder that must be replaced with the actual address by post processing the generated file.

Embedded Target for Infineon C166® Microcontrollers performs this post processing for you. To do this it first extracts the memory address information from the map file generated during the link process. Secondly, it replaces the placeholders in the ASAP2 file with the actual memory addresses. This post processing is performed automatically and requires no additional input from you.

Integrating Your Own Device Drivers

This section includes the following topics:

Integrating Hand-Coded Device Drivers with a Simulink Model (p. 3-2)	Overview of the steps required to integrate your device drivers with a Simulink model
Preparing Input and Output Signals to the Device Driver Functions (p. 3-3)	How to structure your model's inputs and outputs using the demo `c166_user_io.mdl` as an example
Calling the Device Driver Functions from c166_main.c (p. 3-6)	Real-Time Workshop settings to call your hand-coded device driver functions
Adding the I/O Driver Source to the List of Files to Build (p. 3-8)	How to customize the Real-Time Workshop make command to integrate your device driver code
Tutorial: Using the Example Driver Functions (p. 3-9)	A tutorial to show you the example driver functions and how they are integrated with Embedded Target for Infineon C166® Microcontrollers. This includes generating, downloading and running code from the controller subsystem of the `c166_user_io.mdl` demo model.

Integrating Hand-Coded Device Drivers with a Simulink Model

Embedded Target for Infineon C166® Microcontrollers has only a limited set of I/O device driver blocks. This means that for most applications, it will be necessary to write some device driver code by hand.

The approach described here requires the following steps:

1. Identify the model inputs/outputs that must be read from/written to device driver functions.
2. Set the data type and storage class for each input or output signal so that it is compatible with your device driver code.
3. Use the hooks provided in the automatically generated `c166_main.c` to call your device driver initialization, input and output functions.
4. Add your device driver source code to the list of files that must be included in the build process.

Each of these steps is described in the following sections. An example model is provided: `c166_user_io.mdl`.

An alternative approach is to create Simulink I/O blocks that automatically generate the device driver code. This approach may be worth considering if you will frequently need to reconfigure the I/O behavior. If you want to take this alternative approach, you should consult the documentation on S-functions and TLC.

A very useful tool for creating C166 device drivers is the freeware Digital Application Engineer DAvE from Infineon. You can find this at the following URL:

`www.infineon.de/dave`

Using this package along with the hardware User's Manual will greatly ease the task of developing your own device driver code.

Preparing Input and Output Signals to the Device Driver Functions

We recommend you structure your model similarly to `c166_user_io.mdl`. Place the control algorithm that will be targeted onto the C166® microcontroller hardware in a separate subsystem. Prior to generating code, you can run this model in closed-loop simulation; this allows you to validate the correct behavior of your control algorithm before running it in real time.

When structuring your model in this way, you should make sure that all the input and output signals to the control algorithm are channeled through top-level input or output ports in the control algorithm subsystem.

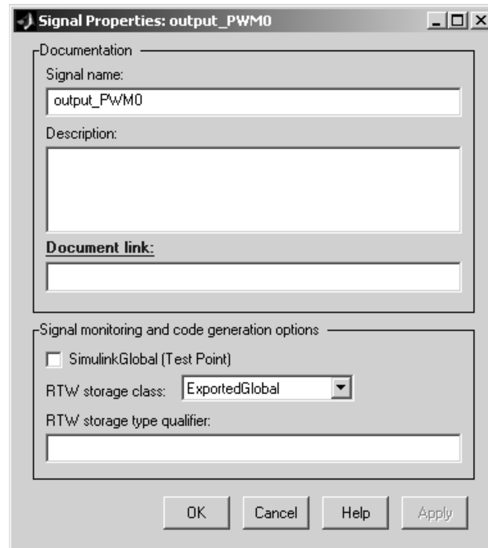
By default, when you generate code for the control algorithm subsystem, Real Time Workshop will choose variable names and data structures for each of the top level input and output signals. However, in this case you must ensure that the variables are global, and that their names and data structures match those that are required by the hand-written device driver functions.

The example model `c166_user_io` illustrates some alternative ways to achieve this. The simplest method is to:

1. Select one of the signals in your model connected to a top level output port in the control algorithm subsystem. As an example, open the demo `c166_user_io.mdl`, open the controller subsystem, and click on the `output_PWM0` signal.

2 Select the menu item **Edit –> Signal Properties**.

The **Signal Properties** dialog appears, as in the example following.



3 Enter the required variable name for your signal in the **Signal name** edit box. This must match the variable name required by your hand written device driver functions.

4 Select `ExportedGlobal` from the **RTW storage class** drop-down menu.

When you generate code for this model, Real-Time Workshop will use the variable name that you have specified and will create an 'extern' declaration in the model header file. By using a `#include` directive to include this model header file in your device driver source code it is possible for the device driver functions to read or write this variable that is defined in the Real-Time Workshop generated code.

A more sophisticated approach is to use custom storage classes. By using custom storage classes you can collect a number of input or output variables together into a C struct, resulting in more readable code. The LED output signal in the `c166_user_io.mdl` uses a custom storage class, which uses a single bit in a bitfield variable. See "Tutorial: Using the Example Driver

Functions" on page 3-9 for details about the different ways the model variables are defined and referenced to interface the hand-coded driver functions and the automatically generated code.

By defining your own custom storage classes, you have complete control over the data structures that are used for any signal in the model. See the custom storage class documentation in the Real-Time Workshop Embedded Coder documentation for more details.

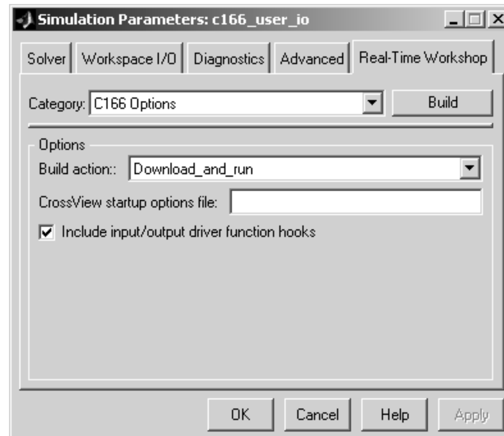
Calling the Device Driver Functions from c166_main.c

You should check the option to include I/O driver function hooks. When Real-Time Workshop generates code for this model, it includes some extra calls to user-supplied I/O device driver functions.

1 Select **Simulation –> Simulation Parameters**.

The **Simulation Parameters** dialog appears.

2 On the Real Time Workshop tab, select `C166 Options` from the **Category** drop-down menu, as shown in the example below.



3 Select the check box option for including I/O driver function hooks.

These functions are

`user_io_initialize` — called following model initialization

`base_rate_model_inputs` — read model inputs, called at the base sample rate

`base_rate_model_outputs` — write model outputs, called at the base sample rate

`sub_rate_i_model_inputs` — read model inputs, called at the start of subrate i, where i=1, 2, ….

sub_rate_i_model_outputs — write model outputs, called at the start of sub-rate i, where i=1, 2, ….

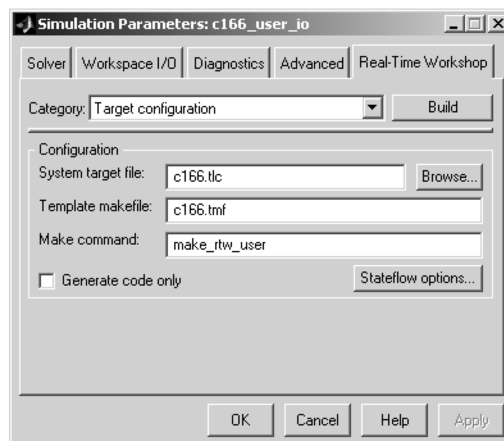
If you are using the automatically generated c166_main.c, then these function names are fixed.

For an example implementation of these functions, open the model c166_user_io and follow the link to open the I/O driver source files. These are described in “Tutorial: Using the Example Driver Functions” on page 3-9.

Adding the I/O Driver Source to the List of Files to Build

You must tell the Real-Time Windows build process to compile and link the I/O driver source files that you have written. To do this, you must add some extra arguments to the `make_rtw` command in the Real-Time Workshop tab of the **Simulation Parameters** dialog.

1 Select `Target configuration` from the **Category** drop-down menu.



2 Alter the **Make command** in the edit box.

If you have several files to add it may be convenient to put the command inside a new file, for example:

`make_rtw_user.m`:

and replace the `make_rtw` command with `make_rtw_user`.

You are now ready to build your model and run it in real time.

You can examine an example of this custom make command in the example model `c166_user_io`. See the instructions in "Tutorial: Using the Example Driver Functions" on page 3-9. Step 8 shows you how to specify the location of your own hand-coded drivers.

Tutorial: Using the Example Driver Functions

The example model `c166_user_io` demonstrates how to integrate user-defined device driver code. In this tutorial you will generate code from the `controller` subsystem, which will automatically download and run on the target.

The model `c166_user_io` illustrates three alternative methods for using global variables to interface the hand-written driver functions with the Real-Time Workshop automatically generated code. The three different methods are illustrated by these signals:

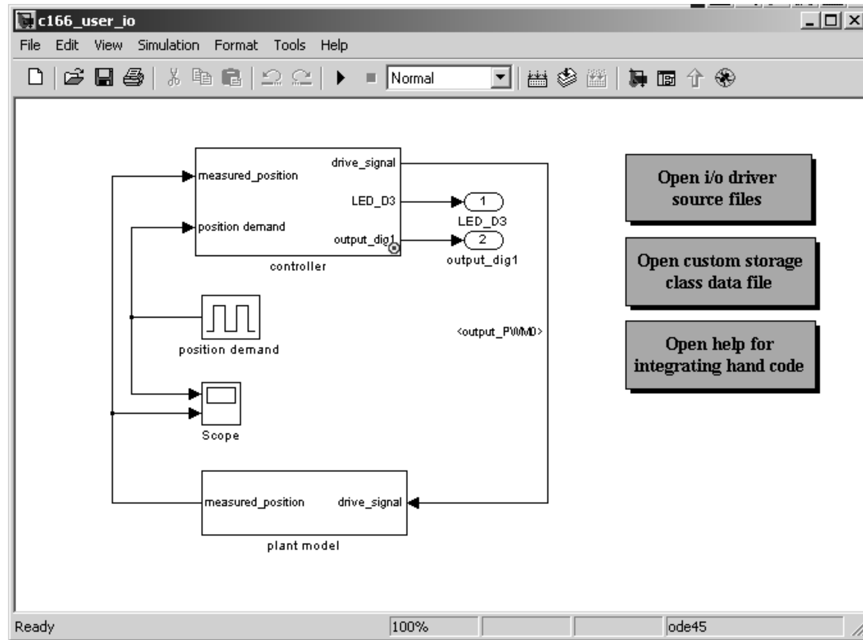
- `input_adc0`
- `output_PWM0`
- `output_led_D3`

For `input_adc0`, the variable is defined in the hand-code and referenced in the Real-Time Workshop code.

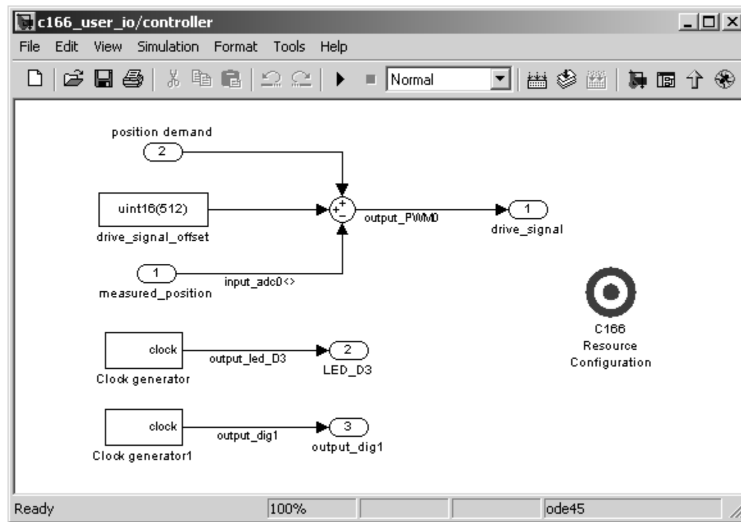
For `output_PWM0` the variable is defined in the Real_time Workshop code and referenced in the hand code.

For `output_led_D3` a more sophisticated approach is used, involving custom storage classes. In this case the variable is again defined in the Real-Time Workshop code and referenced by the hand code; the difference is that the variable is defined and referenced as a bitfield using C166® microcontroller bit-addressable memory.

1 Open the model `c166_user_io.mdl`.

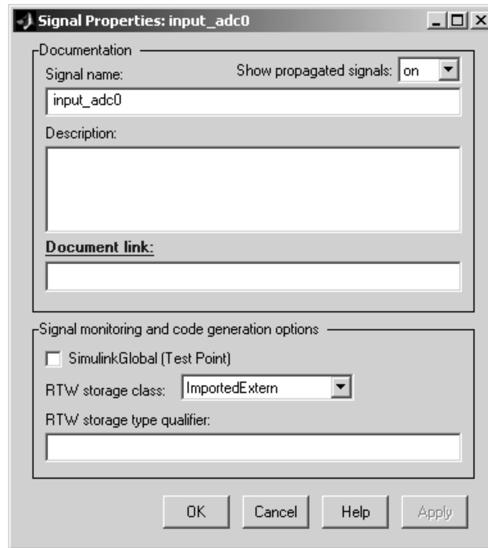


2 Open the `controller` subsystem by double-clicking and select the signal `input_adc0<>`.



3 Select the menu item **Edit –> Signal Properties**.

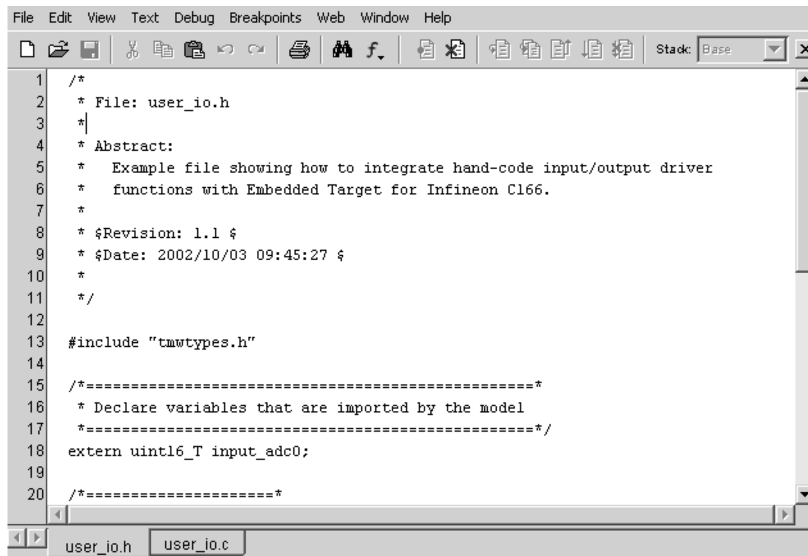
The **Signal Properties** dialog appears.



Observe that the **RTW storage class** is `ImportedExtern`. When you generate code for this model, Real-Time Workshop will use the specified variable name `input_adc0` and will create an `extern` declaration in the model header file. Since the Real-Time Workshop storage class is `ImportedExtern`, this variable must be defined in the hand-written driver code. When you open the file `user_io.c` in the next step, you will find the line `uint16_T input_adc0` that provides this definition.

4 Double-click the link in the top level model **Open the i/o driver source files**.

Two source files open in the MATLAB editor, `user_io.h` and `user_io.c`.

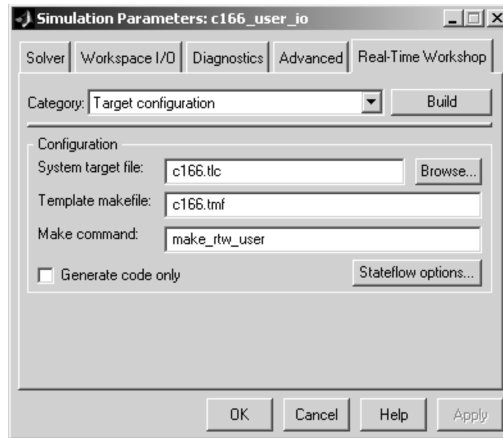


5 Click the `user_io.h` tab, as shown above. Here you can see "`extern uint16_T input_adc0`" under the heading "`Declare variables that are imported by the model`". Also look at the `#include` directive in `user_io.c`. The `extern` declaration and incorporating the header file into the build makes it possible for the device driver functions to read or write this variable that is defined in the Real-Time Workshop generated code.

6 In the `controller` subsystem, select **Simulation** –> **Simulation parameters**.

7 On the Real-Time Workshop tab, look at the **Make command**:

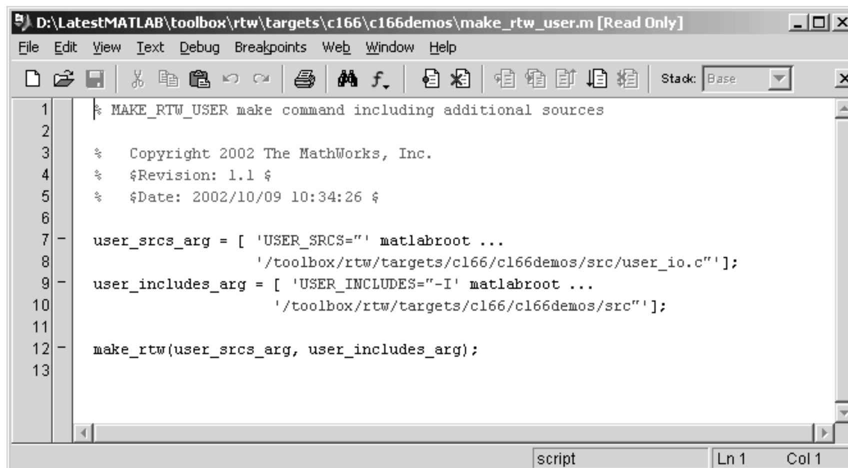
```
make_rtw_user
```



This command instructs Real-Time Workshop to compile and link the hand-coded I/O driver source files specified in the make file in the build process.

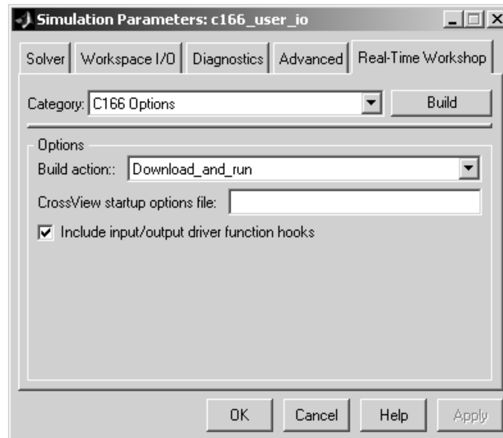
8 Look at the make file to see how these are specified. At the command line type:

```
edit make_rtw_user
```



Observe the lines specifying the path to the hand-coded I/O driver source files to be compiled and linked. This is where you would specify the location of your own hand-coded drivers. For this tutorial do not make changes in the make file. Close the editor and return to the **Simulation Parameters** dialog.

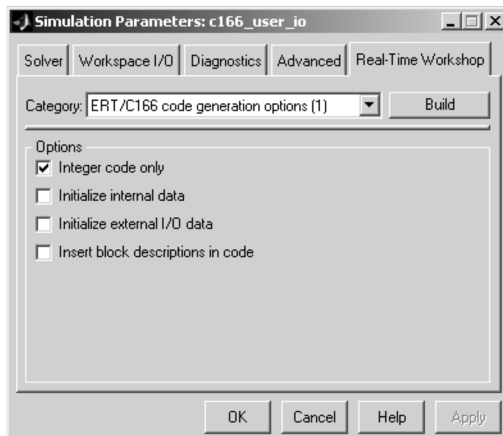
9 Select `C166 Options` from the **Category** drop-down menu. Observe the selected option **Include i/o driver function hooks**.



This instructs Real-Time Workshop to include extra calls to the user-supplied I/O device driver functions when code is generated for this model.

10 Select C166/`ERT_code_generation_options (1)` from the **Category** drop-down menu. Observe the selected option **Integer code only**.

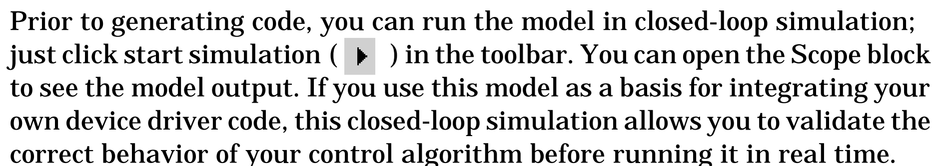
If your model does not use floating point, you should check the option to use integer code only. Using integer code only will result in smaller code size and faster real-time execution. It also speeds up the build process because libraries that are only used by floating-point applications are not included.



Explore the `user_io.c` file. This example file is intended to show you some hand-coded input/output driver functions and how they can be integrated with Embedded Target for Infineon C166® Microcontrollers.

You can see sections for initializing these input/output drivers: ADC, digital i/o, and Pulse Width Modulation (PWM).

11 Close the **Signal Properties** and **Simulation Parameters** dialogs if they are still open by clicking **Cancel**.



Prior to generating code, you can run the model in closed-loop simulation; just click start simulation ( ) in the toolbar. You can open the Scope block to see the model output. If you use this model as a basis for integrating your own device driver code, this closed-loop simulation allows you to validate the correct behavior of your control algorithm before running it in real time.

12 Generate code by right-clicking on the `controller` subsystem and selecting **Real-Time Workshop –> Build Subsystem**.

13 Click **Build** in the **Build code for Subsystem: Controller** dialog which appears. Watch the messages as the process proceeds; code is generated, downloads, and runs on the target.

If you are using a Phytec phyCORE module with HD200 development board, the digital output is connected to the LED D3. You can see successful execution of the code when the LED blinks.

4

Custom Storage Class for C166® Microcontroller Bit-Addressable Memory

This section contains the following topics:

Specifying C166® Microcontroller Bit-Addressable Memory (p. 4-2)	How to use Embedded Target for Infineon C166® Microcontrollers to take advantage of C166® microcontroller bit-addressable memory. This can significantly reduce code size and increase execution speed.
Using the Bitfield Example Model (p. 4-3)	This is a step-by-step guide to the example model `c166_bitfields.mdl`. This model is configured to launch the debugger at the end of the build. Included is a comparison with another custom storage class variable in `c166_user_io.mdl`

Specifying C166® Microcontroller Bit-Addressable Memory

Embedded Target for Infineon C166® Microcontrollers allows you to take advantage of C166® microcontroller bit-addressable memory. The example model `c166_bitfields.mdl` demonstrates this. By using bit-addressable memory, the compiler is able to use special assembler instructions that significantly reduce code size and increase execution speed.

At the Simulink level, this is done by using the custom storage class SimulinkC166.Signal. To specify that a signal in the model should use bit-addressable memory, you must perform the following steps:

1. Ensure that the signal has the Simulink datatype 'boolean'.
2. Attach a label to the signal, either using **Edit –> Signal Properties**, or by double-clicking on the signal and typing in the name directly; this label will be used as the bitfield variable name in the generated code.
3. Create a new Simulink data object of type **SimulinkC166.Signal** with the same name as the signal label. See the file `c166bitfielddata.m` for an example.
4. You can select **Tools –> Data Explorer** to inspect all the Simulink data objects that are available to the model.
5. Build the model.

The example model `c166_bitfields.mdl` is configured to start the debugger at the end of the build. To try this see the next section “Using the Bitfield Example Model” on page 4-3.

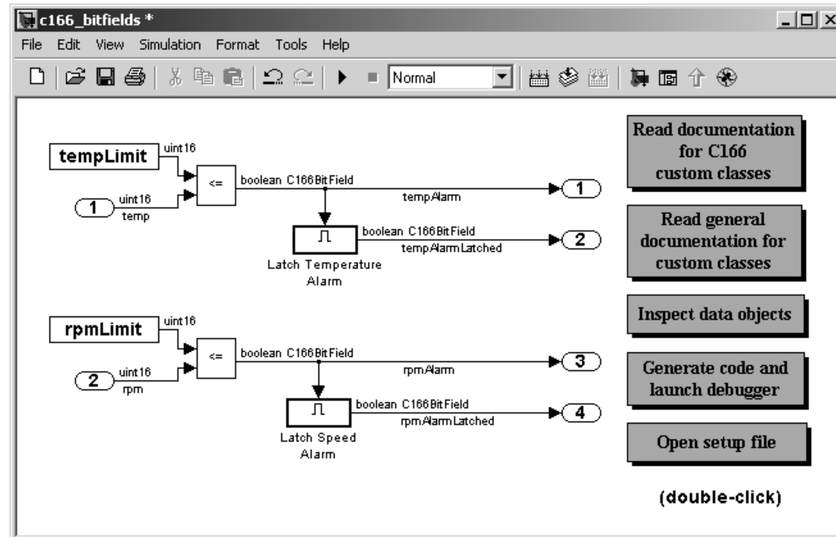
One of the signals in the demo model `c166_user_io.mdl` also uses the custom storage class `SimulinkC166.Signal` to specify that the signal uses bit-addressable memory. You can compare this with the `c166_bitfields` example; it is included in the steps in “Using the Bitfield Example Model” on page 4-3.

Using the Bitfield Example Model

You can use the example model `c166_bitfields.mdl` to see automatic debugger start at the end of the build.

Follow these steps:

1 Open `c166_bitfields.mdl`.

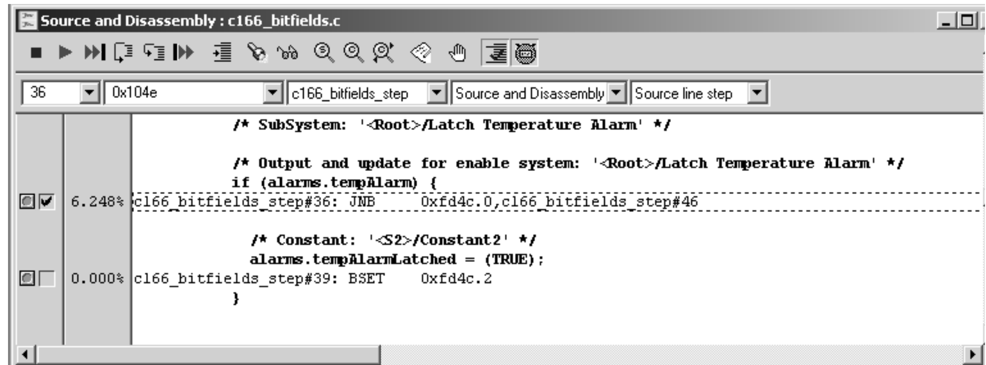


2 Double-click **Generate code and launch debugger**.

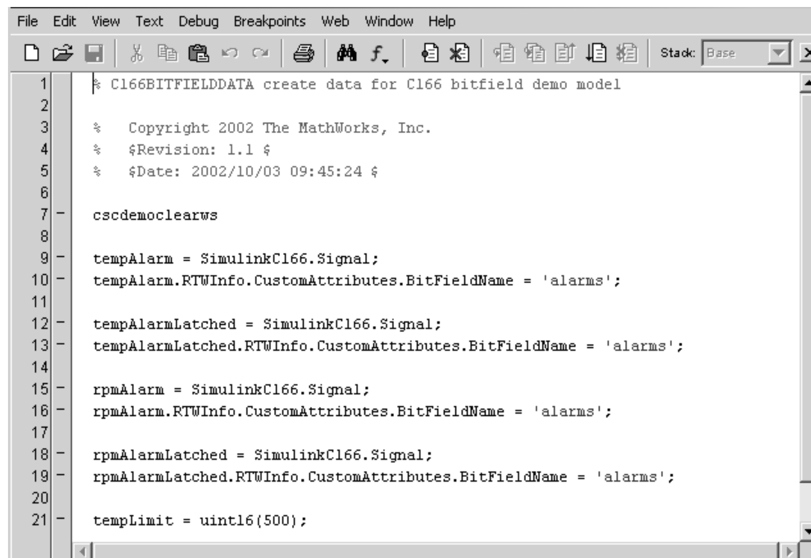
Code is generated and the debugger is started.

3 Select **View –> Source –> Source and Disassembly**.

The example following shows a sample of the generated code.



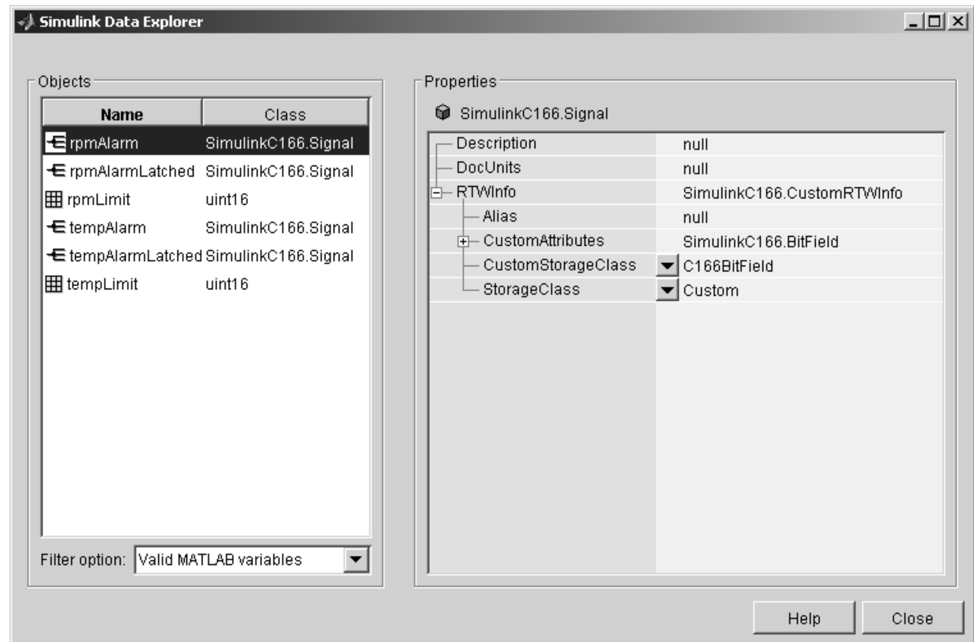
4 You can double-click **Open setup file** in the model to open the file `c166bitfielddata.m` in the MATLAB editor.



This file creates a new Simulink data object using the custom storage class `SimulinkC166.Signal`. By using custom storage classes, you can collect a number of input or output variables together into a C struct, resulting in more readable code. By defining your own custom storage classes, you have complete control over the data structures that are used for any signal in the

model. See the custom storage class documentation in the Real-Time Workshop Embedded Coder User's Guide for more details. You can double click **Read general documentation for custom storage classes** in the model to go directly to the relevent Embedded Coder help section.

5 You can double-click **Inspect data objects** (or select **Tools –> Data Explorer...**) to inspect all the Simulink data objects that are available to the model.

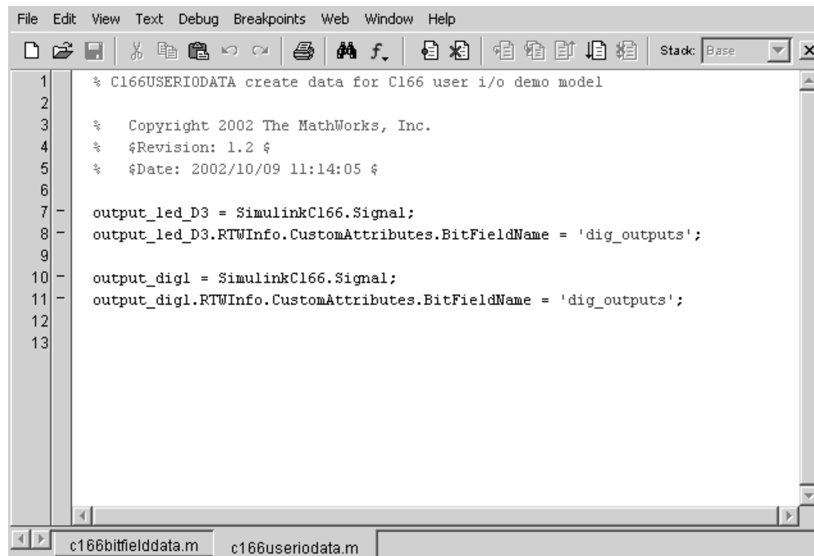


Here you can see the `SimulinkC166.Signal` data object and you can click on each object to inspect the properties.

6 One of the signals in the demo model `c166_user_io` also uses the custom storage class `SimulinkC166.Signal` to specify that the signal uses bit-addressable memory. Open `c166_user_io.mdl`.

7 Double-click **Open custom storage class data file**.

The file `c166useriodata.m` opens in the MATLAB editor.



Compare with `c166bitfielddata.m`.

For more details on the variables in this model see “Tutorial: Using the Example Driver Functions” on page 3–9.

5

Block Reference

This section contains the following topics:

The Embedded Target for Infineon C166® Microcontrollers Block Library (p. 5-2)	Overview of the block libraries provided by the Embedded Target for Infineon C166® Microcontrollers.
Blocks Organized by Library (p. 5-3)	Block summaries and links to the block reference documentation, grouped by block library.
Alphabetical List of Blocks (p. 5-7)	Block summaries and links to the block reference documentation, in alphabetical order.

The Embedded Target for Infineon C166® Microcontrollers Block Library

The Embedded Target for Infineon C166® Microcontrollers provides one block library:

- C166 Driver Library

The following sections provide complete information on each block in the Embedded Target for Infineon C166® Microcontrollers block libraries, in a structured format. Refer to these pages when you need details about a specific block. Click **Help** on the **Block Parameters** dialog for the block, or access the block reference page through Help.

Using Block Reference Pages

Block reference pages are listed in alphabetical order by the block name. Each entry contains the following information:

- **Purpose** — describes why you use the block or function.
- **Library** — identifies the block library where you find the block.
- **Description** — describes what the block does.
- **Dialog Box** — shows the block parameters dialog and describes the parameters and options contained in the dialog. Each parameter or option appears with the appropriate choices and effects.
- **Examples** — optional section that provides demonstration models to highlight block features.

In addition, block reference pages provide pictures of the Simulink model icon for the blocks.

Blocks Organized by Library

The blocks in the Embedded Target for Infineon C166® Microcontrollers are organized into sublibraries that support different functions. The tables below reflect that organization.

C166 Drivers Library

Top Level Library

Block Name	Purpose
C166 Resource Configuration	Support driver configuration for C166® microcontrollers

---

Note To automatically generate code from a 'main' model using the Embedded Target for Infineon C166® Microcontrollers real-time target, a C166 Resource Configuration block must be included in the model.

The C166 Resource Configuration block is only required if you are generating 'main' automatically. It is not required if you are using your own user supplied main. The Resource Configuration block provides information required for generating timer interrupt code. If you are using the automatically generated 'main' but do not include a Resource Configuration block in your model, the code will simply execute as fast as possible. That is, it will not be synchronized to real-time. This behaviour may be desirable if you are running code on the debugger/hardware simulator.

Note When using device driver blocks from the Embedded Target for Infineon C166® Microcontrollers libraries in conjunction with the C166 Resource Configuration block, do not disable or break library links on the driver blocks. If library links are disabled or broken, the C166 Resource Configuration block will operate incorrectly. See "C166 Resource Configuration" on page 5-9 for further information.

Asynchronous/Synchronous Serial Interface Sublibrary

Block Name	Purpose
Serial Transmit	Configures serial output
Serial Receive	Configures serial input

Data Type Support and Scaling for Device Driver Blocks

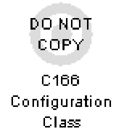
The following table summarizes the input and output data types supported by the device driver blocks in the C166 Drivers library, and the scaling applied to block inputs and outputs.

I/O Data Types and Scaling for C166 Device Driver Blocks

Block	Input Data Type	Input Scaling	Output Data Type	Output Scaling/ Units
Serial Transmit	`uint8`	N/A	`uint8`	N/A
Serial Receive	`uint8`	N/A	`uint8`	N/A

Configuration Class Blocks

Each sublibrary of the Embedded Target for Infineon C166® Microcontrollers library contains a *configuration class block* that has an icon similar to the one shown in this picture.



Note Configuration class blocks exist only to provide information to other blocks. *Do not copy these objects into a model under any circumstances.*

Alphabetical List of Blocks

C166 Resource Configuration 5-8
Serial Transmit 5-15
Serial Receive 5-17

Purpose

Support device configuration for the C166® microcontroller

Library

C166 Drivers

Description

The C166 Resource Configuration block differs in function and behavior from conventional blocks. Therefore, we refer to this block as the C166 Resource Configuration *object*.

The C166 Resource Configuration object is required to provide information that is used to configure driver blocks and timer interrupts.

- You must include this block in your model if:
  - You are using any of the driver blocks supplied with Embedded Target for Infineon C166® Microcontrollers
  - You are taking advantage of the automatically generated scheduler that is driven by timer interrupts.
- You do not need to include the Resource Configuration object in your model if you are not using any of the C166 driver library blocks, and if you do not require the automatically generated scheduler (for example, if you are supplying your own `main.c`).

The C166 Resource Configuration object maintains configuration settings that apply to the C166® microcontroller. Although the C166 Resource Configuration object resembles a conventional block in appearance, it is not connected to other blocks via input or output ports. This is because the purpose of the C166 Resource Configuration object is to provide information to other blocks in the model. C166 device driver blocks register their presence with the C166 Resource Configuration object when they are added to a model or subsystem; they can then query the C166 Resource Configuration object for required information.

To install a C166 Resource Configuration object in a model or subsystem, open the C166 Drivers library and select the C166 Resource Configuration icon. Then drag and drop it into your model or subsystem, like a conventional block.

Having installed a C166 Resource Configuration object into your model or subsystem, you can then select and edit configuration settings in the C166 Resource Configuration window. See "Using the C166 Resource Configuration Window" on page 5-11 for further information.

Note If your model or subsystem requires a C166 Resource Configuration object (see above), you should place it at the top level system for which you are going to generate code. If your whole model is going to run on the target processor, put the C166 Resource Configuration object at the root level of the model. If you are going to generate code from separate subsystems (to run specific subsystems on the target), place a C166 Resource Configuration object at the top level of each subsystem. You should not have more than one C166 Resource Configuration object in the same branch of the model hierarchy. Errors will result if these conditions are not met.

Types of Configurations

A *configuration* is a collection of parameter values affecting the operation of one or more device driver blocks in the Embedded Target for Infineon C166® Microcontrollers library. The C166 Resource Configuration object currently supports the following types of configurations:

- C166 Drivers Configuration: C166® microcontroller clocks and other CPU-related parameters.
- Asynchronous/Synchronous serial Interface Configuration: parameters related to the serial driver blocks and Simulink external mode.

Dialog Box

The C166 Drivers Configuration always appears in the active configuration pane. If there are also blocks in your model from the Asynchronous/Synchronous Serial Interface (ASC0) sublibrary, you will also see the configuration for these, as seen in the next example. If you add an ASC0 block to a model without any ASC0 blocks, the appropriate configuration is created and activated in the C166 Resource Configuration block.

You can see an example like this by opening the demo model `c166_serial_transmit` and double-clicking on the C166 Resource Configuration block.

A configuration remains active until all blocks associated with it are removed from the model or subsystem. At that point, the configuration is in an *inactive* state. Inactive configurations are not shown in the C166 Resource

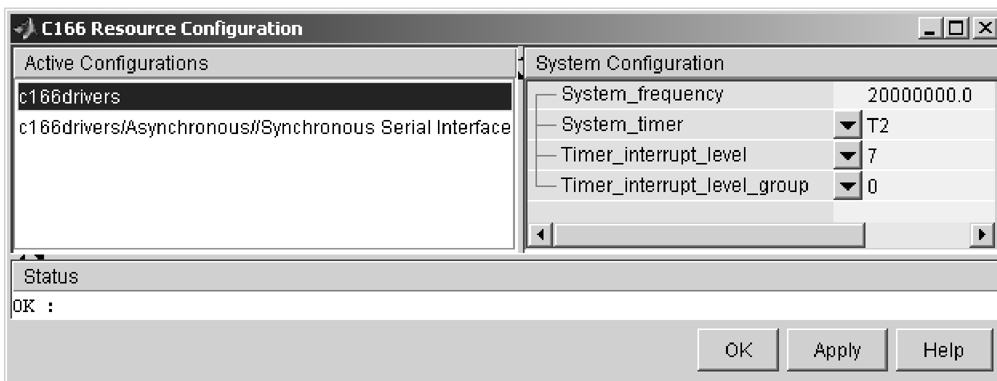
Configuration window. You can reactivate a configuration by simply adding an appropriate block into the model.

When you save a model that contains inactive configurations, you have the option to either save inactive configurations with the model, or delete them.

Using the C166 Resource Configuration Window

To open the C166 Resource Configuration window, install a C166 Resource Configuration object in your model or subsystem, and double-click on the C166 Resource Configuration icon. The C166 Resource Configuration window then opens.

This example shows the C166 Resource Configuration window for a model that has active configurations for the C166® microcontroller (`c166drivers`) and for the Asynchronous/Synchronous Serial Interface (ASC0) blocks, as found in the demo `c166_serial_transmit`.



The C166 Resource Configuration window consists of the following elements:

- **Active Configurations** panel: This panel displays a list of currently active configurations. To edit a configuration, click on its entry in the list. The parameters for the selected configuration then appear in the **System configuration** panel.

  To link back to the library associated with an active configuration, right-click on its entry in the list. From the pop-up menu that appears, select **Go to library**.

  To see documentation associated with an active configuration, right-click on its entry in the list. From the pop-up menu that appears, select **Help**.
- **System configuration** panel: This panel lets you edit the parameters of the selected configuration. The parameters of each configuration type are detailed in "C166 Resource Configuration Window Parameters" on page 5-12.

Note Click **Apply** to make your changes take effect.

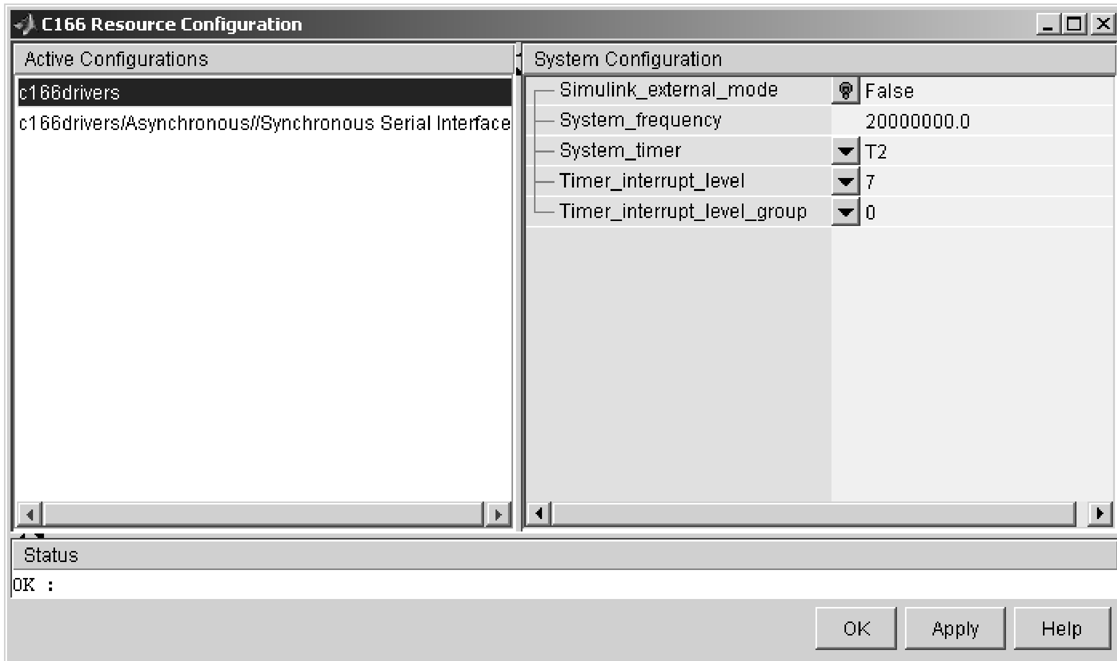
- **Status** panel: The **Status** panel displays error messages that may arise if resource allocation conflicts are detected in the configuration.
- **OK** button: Dismisses the window.

C166 Resource Configuration Window Parameters

The sections following describe the parameters for each type of configuration in the C166 Resource Configuration window. The default parameter settings are optimal for most purposes. If you want to change the settings, we suggest you read the sections of the *C166 Users Manual* referenced below. You can find this document at the Infineon web site at the following URL:

`http://www.infineon.com/`

C166 System Configuration Parameters



System_frequency

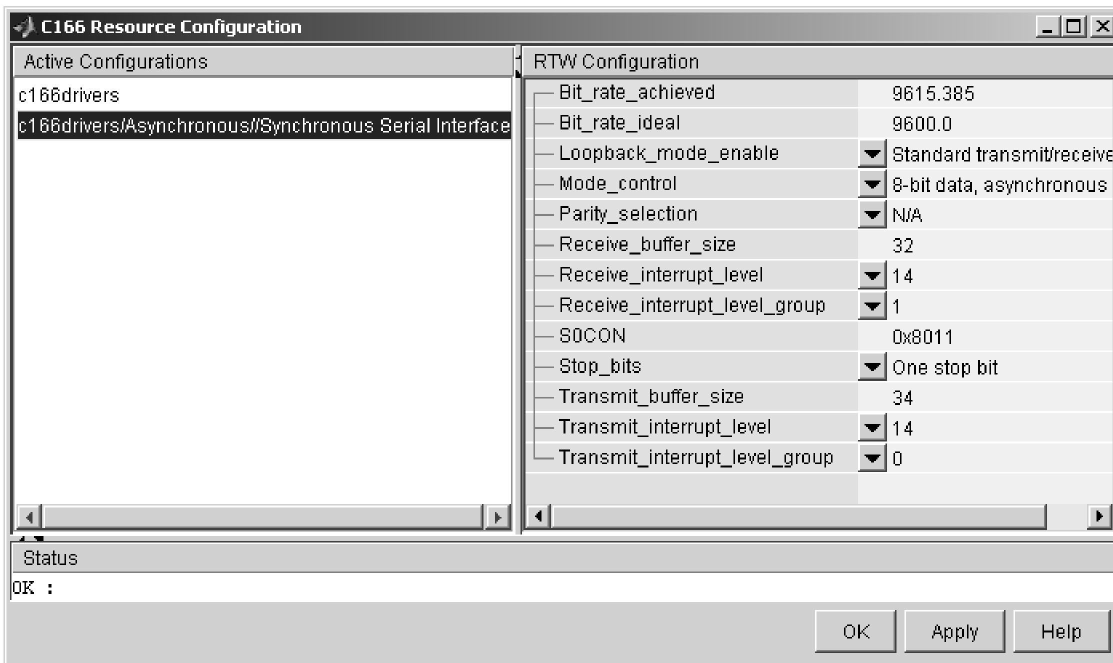
You must set the system frequency of your C166® microcontroller hardware here. Note that the value will depend on your hardware type and configuration. Should you choose an incorrect value this will cause the model to be correspondingly fast or slow.

System_timer

You must select which timer to use for generating interrupts to drive the model update rate. You should select a timer, or timer pair, that you do not intend to use for any other purpose within your application. We recommend you choose a pair of timers, e.g. `T6, with reload from CAPREL`; this will give the best possible sample time accuracy and there will be no long term drift caused by higher priority interrupts. If you choose a single timer, e.g., `T2`, the timer value will be reloaded within the timer interrupt service routine. With this approach any delay in servicing the timer interrupt will be added to the time until the next timer interrupt is generated.

Timer_interrupt_level and **Timer_interrupt_level_group**
These two parameters together set the priority of sample time interrupts. You should choose values such that the sample time interrupts are suitably prioritized relative to other interrupts used by your application.

Asynchronous/Synchronous Serial Interface Configuration Parameters



Bit_rate_achieved
This read-only field shows the achieved serial interface bit rate. In general this value differs slightly from the requested bit rate, but is the closest value that can be achieved by setting allowed values in C166 register S0BG and bitfield S0BRS of register S0CON.

Bit_rate_ideal

Enter the desired bit rate for serial communications in this field. Appropriate register settings will be calculated automatically. You can verify the actual bit rate in the `Bit_rate_achieved` field.

Loopback_mode_enable

Select this entry to operate the serial interface in loopback mode. This may be useful for test purposes where the serial interface is required to receive data that it transmitted itself.

Mode_control

Select the desired combination of word length and parity/no parity. See the C166® Microcontroller User's Manual for more details.

Parity_selection

If parity is enabled, you must select odd or even.

Receive_buffer_size

You must select the size of the RAM buffer that will be used by the serial receive driver. The maximum allowed value is 254.

Receive_interrupt_level and **Receive_interrupt_level_group**

Set the receive interrupt priority here. Note that the drivers used by Embedded Target for Infineon C166® Microcontrollers only allow interrupt levels 14 and 15 to be used. The reason for this is that the drivers use the PEC (peripheral event controller), which provides very fast interrupt response but is restricted to levels 14 and 15.

S0CON

This is a noneditable field that shows the value of the serial interface register S0CON and how it varies as dialog settings are changed.

Stop_bits

You must select either 1 or 2 stop bits.

Transmit_buffer_size

See `Receive_buffer_size`.

Transmit_interrupt_level and **Transmit_interrupt_level_group**

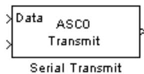
See `Receive` parameters above.

Serial Transmit

Purpose Configures C166® microcontroller for serial transmit.

Library Asynchronous/Synchronous Serial Interface

Description The Serial Transmit block transmits bytes over the C166® microcontroller Synchronous/Asynchronous Serial Interface ASC0. You can use it either to transmit a fixed number of bytes, or by enabling the second input, transmit a variable number of bytes each time this block is called.



When the block is called, the specified number of bytes are placed in a FIFO buffer that is internal to the device driver. If this buffer is already full, or if the number of spaces available is too few then not all of the bytes requested will actually be queued for transmit; in this case the number of bytes actually transmitted can be determined from block output.

Once bytes are queued for transmit, they will be sent as fast as possible by the serial interface hardware with no further intervention required by the main application. Note that after each byte is sent a Peripheral Event Controller (PEC) interrupt is generated to fetch the next byte from the internal buffer. The PEC interrupts are extremely fast and have minimal effect on the rest of the application.

To configure the serial interface bit rate, buffer size, PEC interrupt priority and other parameters see the “Asynchronous/Synchronous Serial Interface Configuration Parameters” on page 5-14.

Note If your model contains a serial transmit or receive block, it is not possible to perform on-chip debugging over the same serial interface. Attempting to use the debugger in this case will cause an error. If you need to debug an application that includes the serial transmit and receive blocks, you must run the debugger using a hardware simulator; alternatively, it may be possible to run your debugger on-chip without using the serial interface, for example if debugging over CAN is available. See “Starting the Debugger on Completion of the Build Process” on page 2-13.

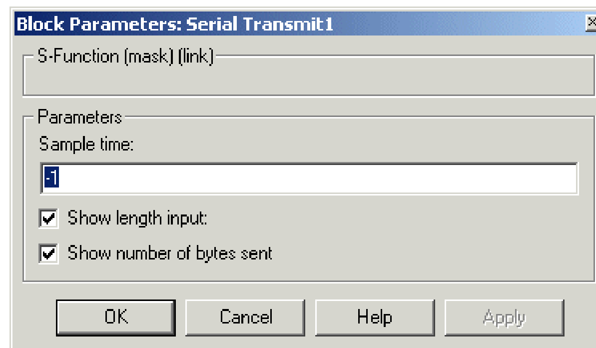
Block Inputs and Outputs

The first input contains the data to be transmitted; this input signal may be either a vector or scalar with data type `uint8`.

The optional second input must be a scalar and may be used to control the number of bytes transmitted. The number of bytes to transmit should not be greater than the width of the first input signal.

The block output port `actual number of bytes output` gives the number of bytes queued for transmit. If there was sufficient space in the buffer, this number will be equal to the requested number of bytes to transmit.

Dialog Box



Show length input

Enable/disable the number of bytes to send. If not selected, the number of bytes sent is just the width of the first inport; if selected, the second input is enabled, which controls the number of bytes to send.

Show number of bytes sent

Enable/disable the number of bytes actually sent. If selected, this value is available from the first output.

Sample time

The time interval between samples. To inherit the sample time, leave this parameter at the default `-1`. See Specifying Sample Time in the Simulink documentation for more information.

Serial Receive

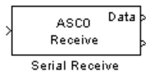
Purpose

Configures C166® microcontroller for serial receive.

Library

Asynchronous/Synchronous Serial Interface

Description



The Serial Receive block receives bytes over the C166® microcontroller Synchronous/Asynchronous Serial Interface ASC0. It requests either a fixed number of bytes to be received, or by enabling the first input, a variable number of bytes can be requested each time this block is called.

When the block is called, the requested number of bytes are retrieved from a FIFO buffer that is internal to the device driver. If this buffer contains fewer bytes than the number requested, these bytes are pulled from the buffer and made available at the block output. The number of bytes actually retrieved from the buffer is made available at the second output. This block will only retrieve bytes that have already been received and placed in the internal buffer; it will never wait for additional data to be received.

Whenever bytes are received at the serial interface, a Peripheral Event Controller (PEC) interrupt is generated to move the byte into the internal buffer. If there is no more space available in the internal buffer, any additional data is lost. The PEC interrupts are extremely fast and have minimal effect on the rest of the application.

To configure the serial interface bit rate, buffer size, PEC interrupt priority and other parameters see the "Asynchronous/Synchronous Serial Interface Configuration Parameters" on page 5-14.

Note If your model contains a serial transmit or receive block, it is not possible to perform on-chip debugging over the same serial interface. Attempting to use the debugger in this case will cause an error. If you need to debug an application that includes the serial transmit and receive blocks, you must run the debugger using a hardware simulator; alternatively, it may be possible to run your debugger on-chip without using the serial interface, for example if debugging over CAN is available. See "Starting the Debugger on Completion of the Build Process" on page 2-13.

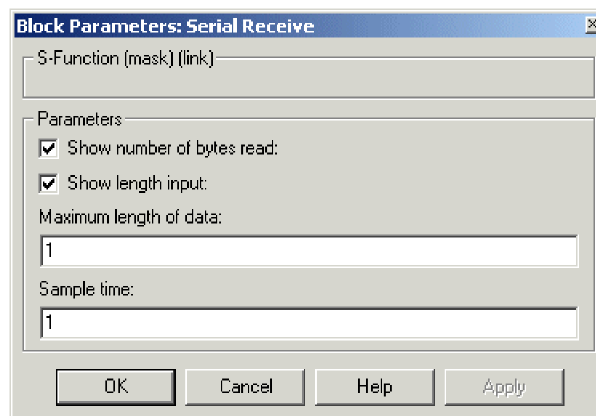
Block Inputs and Outputs

The input can be enabled so a variable number of bytes can be requested each time.

The first output pulls bytes from the buffer – either the number requested or the number available, whichever is the lower. Note that the number requested is value of input signal if supplied, or width of output signal otherwise.

The second output is the number of bytes actually retrieved from the buffer.

Dialog Box



Show number of bytes read

Enables second output to show actual number of bytes retrieved from the buffer.

Show length input

Enables inport so you can vary the number of bytes requested per call.

Maximum length of data

Set this as required up to the maximum buffer size. You can set receive and transmit buffer size (up to a maximum of 256 bytes) within the C166 Resource Configuration object. See “Asynchronous/Synchronous Serial Interface Configuration Parameters” on page 5-14.

Sample time

The time interval between samples. The default is 1. To inherit the sample time, set this parameter to -1. See Specifying Sample Time in the Simulink documentation for more information.

A
ASAP2 files
    generating for C166 2-15
C
Configuration Class blocks 5-5
D
device driver blocks
    C166 Resource Configuration 5-8
    C166 Serial Receive 5-17
    C166 Serial Transmit 5-15
    input data types 5-4
    input scaling 5-4
    output data types 5-4
    output scaling 5-4
E
example model
    c166_bitfields 4-1
    c166_fuelsys 2-15
    c166_serial_io 2-11
    c166_serial_transmit 2-3
    c166_user_io 3-1
I
installation of Embedded Target for Infineon C166 x
R
real-time target
    C166 tutorial 2-3
T
typographical conventions (table) x