# Embedded Target for Infineon C166®
# Microcontrollers

**For Use with Real-Time Workshop®**

■ Modeling

■ Simulation

■ Implementation

## User's Guide

*Version 1*

The MathWorks

**How to Contact The MathWorks**

| | |
|---|---|
| www.mathworks.com | Web |
| comp.soft-sys.matlab | Newsgroup |
| www.mathworks.com/contact_TS.html | 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.
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 2002–2006 by The MathWorks, Inc.

**Trademarks**

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks, and SimBiology, SimEvents, and SimHydraulics are trademarks of The MathWorks, Inc.

C166 is a registered trademark of Infineon Technologies AG.

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

**Patents**

The MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

**Revision History**

# Contents

## Tutorial: Simple Example Applications for C166 Microcontrollers

**2**

## Integrating Your Own Device Drivers

**3**

# Custom Storage Class for C166 Microcontroller Bit-Addressable Memory

**4**

# Execution Profiling

**5**

# **Blocks — By Category**

## **6**

# **Blocks — Alphabetical List**

## **7**

# **Index**

# Getting Started

This section contains the following topics:

# What Is the Embedded Target for Infineon C166 Microcontrollers?

The Embedded Target for Infineon C166® Microcontrollers is an add-on product for use with the Link for TASKING® and 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 XC16x, and ST Microelectronics ST10 (`http://www.us.st.com`).

Used in conjunction with Simulink®, Stateflow®, and the Link for TASKING, 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 and Processor-in-the-Loop (PIL) 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 the `main` program including singletasking or preemptive multitasking scheduler
- Scheduler is configurable to allow temporary overruns
- 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
- Driver blocks for CAN message transmit and receive
- Task execution time profiling
- Examples to show you how to integrate your own driver code

- Fully integrated with Tasking toolchain, using the Link for TASKING project-based process

- 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

- Support for CAN Calibration Protocol

# Prerequisites

---

**Note** You should familiarize yourself with the Link for TASKING documentation, especially "Getting Started".

---

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 "Getting Started with Real-Time Workshop" section of the Real-Time Workshop documentation:

- "What Is Real-Time Workshop?" This section introduces general concepts and terminology related to Real Time Workshop.

- "Working with Real-Time Workshop" This section provides several hands-on exercises that demonstrate the Real-Time Workshop user interface, code generation and build process, and other essential features.

In addition, if you want to understand and use the device driver blocks in the Embedded Target for Infineon C166 Microcontrollers library, you should have at least a basic understanding of the architecture of the C166. The C166 User's Manual (or corresponding document for your C166 derivative processor) is required reading. The MathWorks recommends 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/
```

# Using This Guide

Follow this 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 in its entirety, paying particular attention to "Setting Up and Verifying Your Installation" on page 1-12.

- 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 Chapter 4, "Custom Storage Class for C166 Microcontroller Bit-Addressable Memory" 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 6, "Blocks — By Category" It is particularly important to read C166 Resource Configuration, as the C166 Resource Configuration block is required to use the device driver blocks.

- To browse the demos available, select **Start > Simulink > Embedded Target for Infineon C166 Microcontrollers > Demos**.

We recommend you work through the tutorials in this User's Guide with step-by-step instructions for using and understanding these demos.

# 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 where you can select which products to install.

# Hardware and Software Requirements

## Host Platform

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

You can see the system requirements for MATLAB online at

```
http://www.mathworks.com/products/system.shtml/Windows
```

## 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.

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

- Phytec phyCORE-167 ST10F269
- Phytec phyCORE-167 C167CS
- Phytec kitCON-167 C167CR
- Infineon XC167CI Starter Kit

You can switch easily between these configurations. For other hardware variants, you will need to change the default configuration settings. For details see "Switching Between Hardware Variants" on page 1-10.

For other hardware variants you need to create a new Link for TASKING template project file. If the processor variant selected in the project is not on the list above then you need to configure the code generation process.

This guide assumes that you are working with the Phytec phyCORE-167CS development board, and documents 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

### Required and Related MathWorks Products

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

- MATLAB
- Simulink
- Real-Time Workshop
- Real-Time Workshop Embedded Coder
- Link for TASKING

Simulink Fixed Point is strongly recommended but not essential, except for one of the demos: `c166_fuelsys`.

For more information about any of these products, see either

- The online documentation for that product, if it is installed
- The MathWorks Web site, at `http://www.mathworks.com/products/target_c166/`

### Supported Cross-Development Tools

In addition to the required MathWorks software, a supported cross-development environment is required.

- See "Supported TASKING Toolsets" in the Link for TASKING documentation for the currently supported cross-development tools for use with the Embedded Target for Infineon C166 Microcontrollers:

**Note** The demo version of the Tasking Cross-Compiler is not supported.

- MiniMon freeware download and monitor utility (version 2.2.3)

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 1-12.

## Switching Between Hardware Variants

There are many different members of the C166 microcontroller family, e.g., C167CS, ST10, XC167CI. For each of these processors, it is appropriate to use different compiler switches and link libraries. Even if you are working with a single processor variant, you may need to build for different memory configurations, for example, depending on whether the application will run from RAM or flash memory. The compilation settings are captured in the project file.

The Embedded Target for Infineon C166 Microcontrollers is supplied with preconfigured projects for targeting the hardware and simulator for the following processor variants:

- Phytec phyCORE-C167CS

- Infineon XC167CI Starter Kit

- Phytec phyCORE-ST10F269

- Phytec kitCON-C167CR

If your hardware variant is not on this list, then you need to create a new Link for TASKING template project (see "Tutorial: Creating New Template Projects") and set the C166 code generation options (see "Code Generation Configuration for Nondefault Processors" on page 1-19).

When switching between target configurations, you should review your Link for TASKING option set and ensure that options are set appropriately for the new configuration.

Additionally, for each model that you build, you must check, and, if necessary, change the following settings in the C166 Resource Configuration block:

- System_frequency

- `External_oscillator_frequency`

To determine the correct value of these parameters, consult your hardware documentation.

It is possible to make all the required changes programmatically: a convenience function `c166switchconfig` is provided for this purpose. This function can be run by double-clicking the block Switch Target Processor Variant inside any of the demo models.

# 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 1-16
- "Setting Target Preferences" on page 1-17

Install the Tasking C Cross-Compiler and CrossView Pro Debugger by following the instructions provided by Altium Limited.

If the CrossView connection to your target hardware requires a serial connection, install the MiniMon download utility. By using MiniMon instead of CrossView to launch your application, the serial connection will be available for other purposes, if required. If your CrossView connection is via a debug interface (for example, on XC16x hardware) then it is not necessary to install MiniMon.

At the time of writing, you can obtain the MiniMon download utility for monitoring the serial interface from the Infineon Web site at this URL:

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

To download the MiniMon utility:

**1** Go to the Infineon Web site, and click the `sitemap`.

**2** Select Product Categories > Microcontrollers > Development Tools and Software -> C166/XC166 Development Tools and Software > Software Downloads.

Find MiniMon in the table, and download and install `Minimon_V2228b.exe`.

Minimon may need to be configured for your target processor.

After you install, you must specify the location of MiniMon in the BootstrapLoaderExe target preference, as detailed in "Setting Target Preferences" on page 1-17. Check that MiniMon is correctly configured for your target, as detailed in the next section.

## Verifying MiniMon Settings

You must check that MiniMon has the correct target settings. Start

MiniMon, then click Configure Hardware (  ) in the toolbar (or select **Target > Configuration**) and make sure the settings are as in the following illustrations.

In general, you should choose configuration settings that are consistent with the values specified in the Tasking EDE project.

Select **Settings > Interface** and ensure that the settings for the serial interface match those in the Resource Configuration block of your model.

To set up a configuration for a C167CR:

**1** Select C167CR from the **Controller type** drop-down list.

**2** Click **Yes** three times when prompted by the dialog boxes asking the following questions:

   **a** Do you want to load default memory units for this Type?

   **b** Do you want to activate the default kernel for this Type?

   **c** Do you want to load default initialization registers of this Type?

**3** Perform the following steps on the Initialize register settings:

   **a** Set SYSCON to 0085.

   **b** Set BUSCON1 to 049F.

   **c** Set ADDRSEL1 to 0006.

   **d** Clear all the other check boxes.

   The register settings should look as shown.
This configuration has been verified with a Phytec kc167 (C167CR).
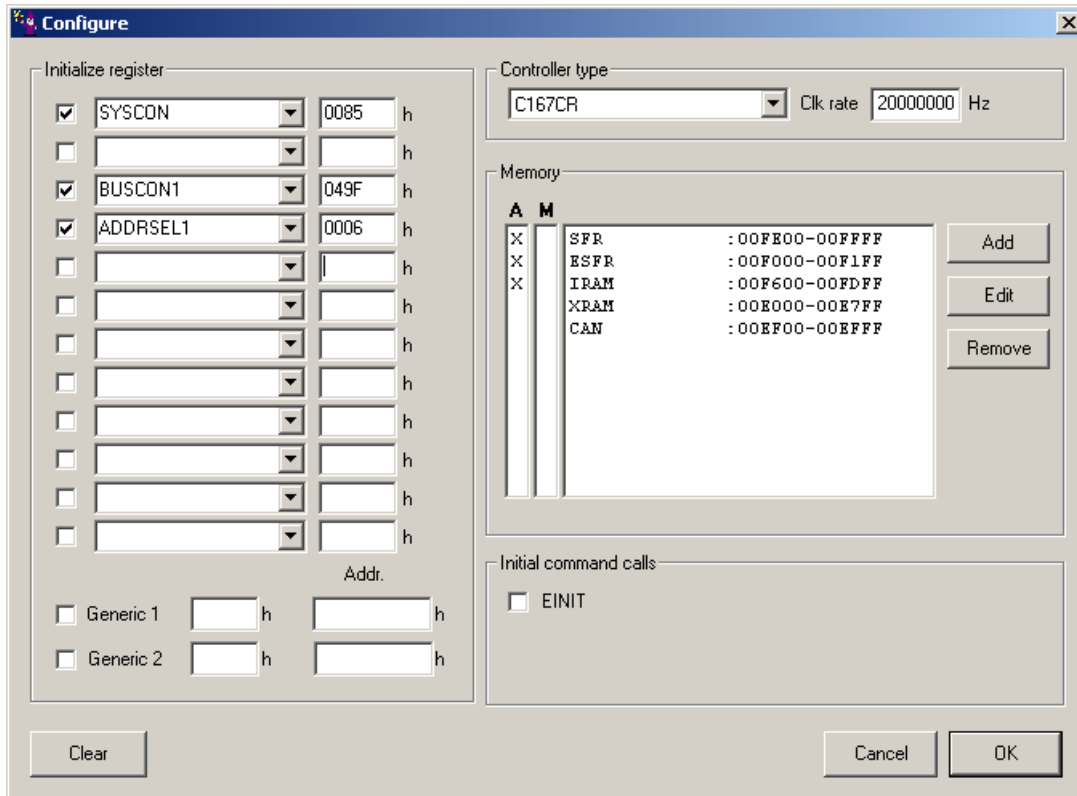
To set up a configuration for a C167CS:

**1** Select C167CS-4RM from the **Controller type** drop-down list.

**2** Click **Yes** three times when prompted by the dialog boxes asking the following questions:

  **a** Do you want to load default memory units for this Type?

  **b** Do you want to activate the default kernel for this Type?

  **c** Do you want to load default initialization registers of this Type?

**3** Perform the following steps on the Initialize register settings:

  **a** Change SYSCON to XPERCON, and set the value to 0403.

**b** Change SYSCON1 to SYSCON, and set the value to 0085.

The order is important: XPERCON must be above SYSCON.

**c** Set BUSCON1 to 049F.

**d** Set ADDRSEL1 to 0006.

**e** Clear all the other check boxes.

The register settings should look as shown.

This second configuration has been verified with phyCORE-C167CS and on phyCORE-ST10F269 hardware.

# Setting Up Your Target Hardware

This guide assumes 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. If you are using different target hardware, you should consult the hardware documentation.

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 these settings can be markedly different from the configuration fresh out of the box.

**2** If you are running applications from RAM only, 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.

1 First you must set up your Link for TASKING target preferences to specify the location of your cross-compiler and other settings. See "Setting Target Preferences" in the Link for TASKING documentation.

2 Start the Embedded Target for Infineon C166 Microcontrollers Target Preferences Setup GUI by selecting **Start > Simulink > Embedded Target for Infineon C166 Microcontrollers > C166 Target Preferences** .
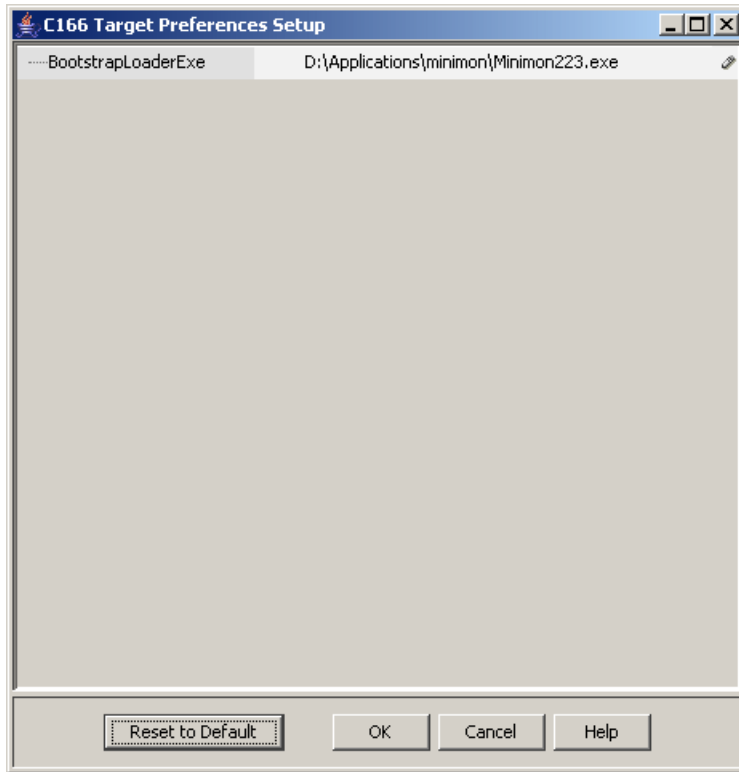
**3** Edit the settings for your cross-development environment:

- BootstrapLoaderExe specifies the path to your download utility (MiniMon).

You must check this path and also verify that the Link for TASKING target preferences 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).

# Code Generation Configuration for Nondefault Processors

If you wish to target nondefault processor types, then you need to set some code generation options in the **TLC Options** of your model's configuration parameters.

If you are using a template that specifies a nondefault processor type, when you try to build the model, you see a build error message similar to the one in the following figure.
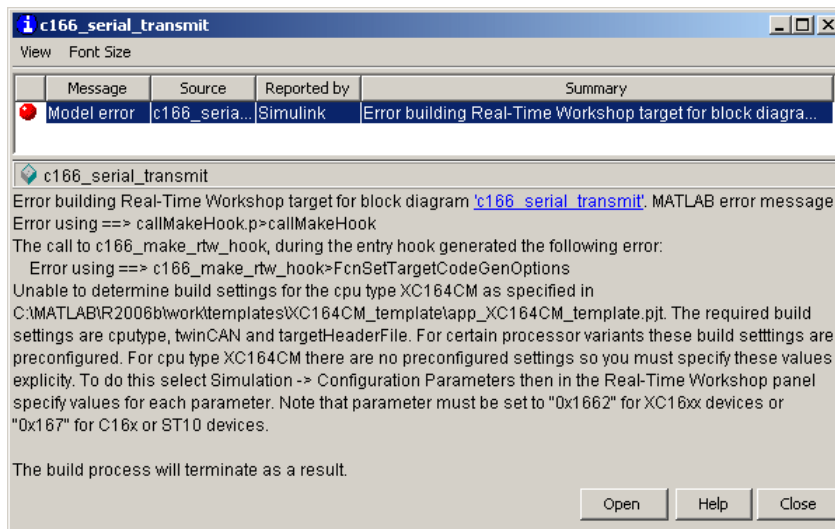


When you open the Configuration Parameters dialog, the parameters you need to set now appear in the TLC Options field. You must replace the string <ENTER VALUE> for each of the parameters cpuType, twinCAN, and targetHeaderFile. The following example shows these parameters before the strings <ENTER VALUE> are replaced.
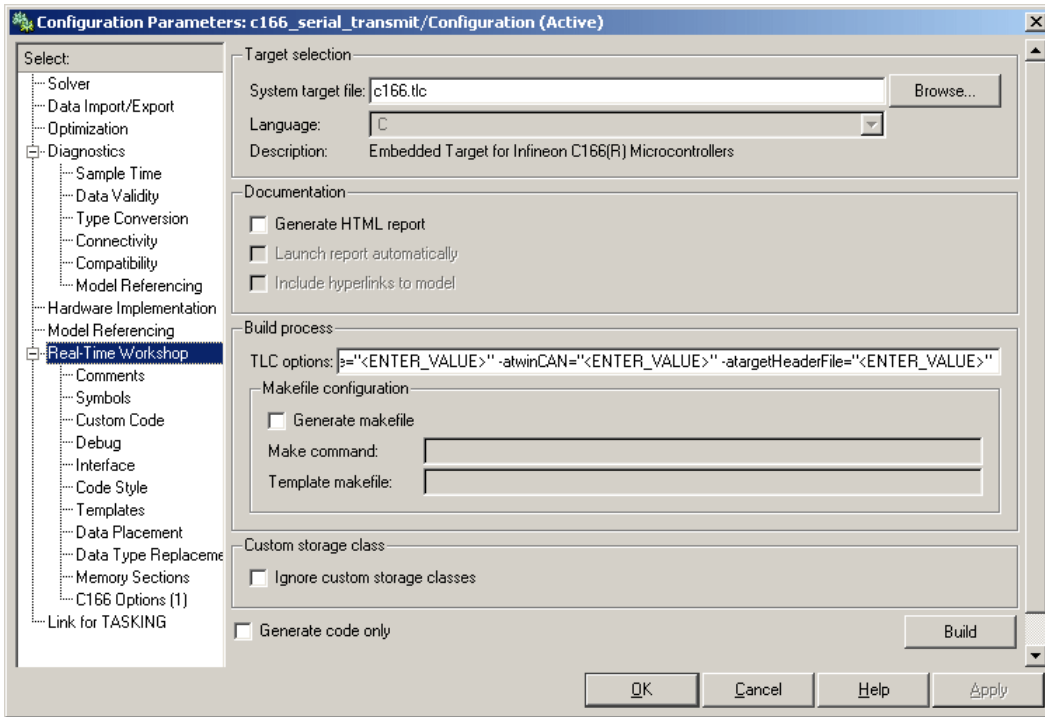
An example configured for an XC164CM is shown in the following figure.
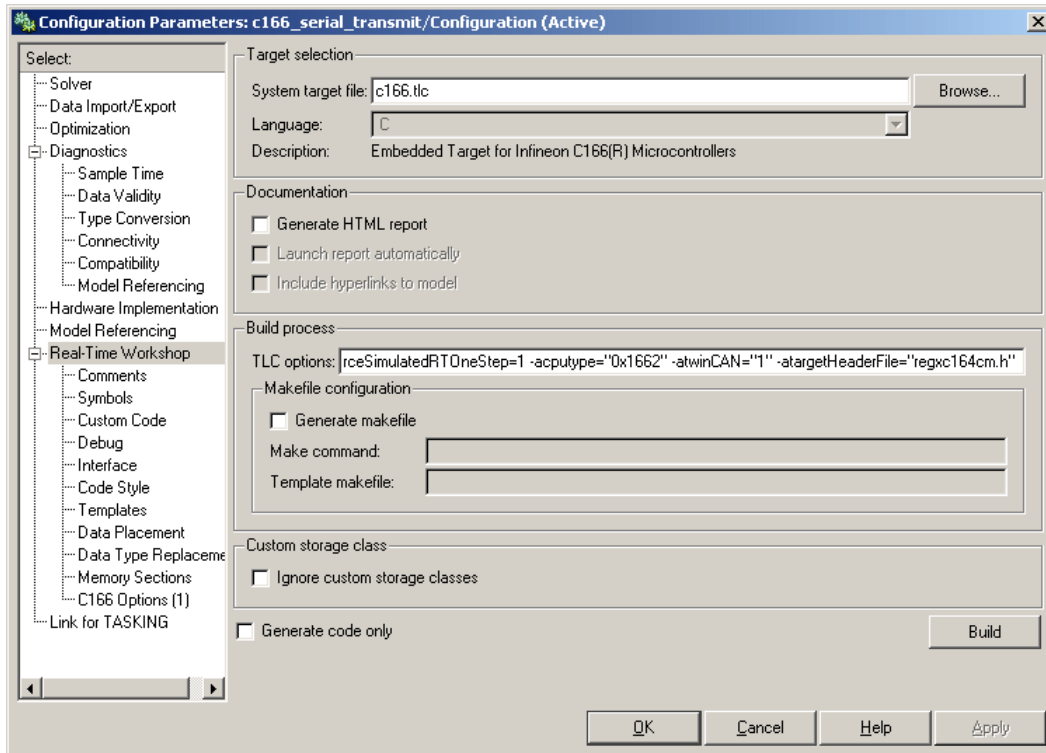
**Summary of Parameters**

The **TLC Options** edit box includes the following parameters:

cpuType

- 0x167, for C16x and ST10 type processors
- 0x1662, for XC16x type processors

twinCAN

- 0 – disabled, for use with processors without TwinCAN support
- 1 – enabled, for use with processors with TwinCAN support

`targetHeaderFile` — The file name of the header file for your processor type. These are found in *TASKING ROOT*\include directory.

**Typical Parameter Configuration**

The following table shows a configuration matrix for the parameters `cpuType`, `twinCAN` and the typical configurations used for the processor variants supported by the Embedded Target.

| Processor Type | CPU type | TwinCAN |
|---|---|---|
| 16x, ST | 0x167 | 0 - disabled |
| XC | 0x1662 | 1 – enabled |

**Note** Driver blocks may not work on unsupported processors.

# 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

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

- 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.

The Embedded Target for Infineon C166 Microcontrollers provides one block library, containing seven sublibraries that support different functions, as follows:

- C166 Drivers Library

  - Asynchronous/Synchronous Serial Interface Sublibrary

  - CAN Interface Sublibrary

  - Execution Profiling Sublibrary

  - TwinCAN Interface Sublibrary

  - Interrupts Sublibrary

  - Utilities Sublibrary

  - Digital Input/Output Sublibrary

See Chapter 6, "Blocks — By Category" for details of each block. You can click **Help** on the Block Parameters dialog box for the block or access the block reference page through Help.

The top-level C166 Drivers library contains the C166 Resource Configuration block. This block supports driver configuration for C166 microcontrollers and is required if there are device driver blocks in the model. See C166 Resource Configuration.

The C166 Resource Configuration block provides information required for generating timer interrupt code. If you do not include a C166 Resource Configuration block in your model, the code simply executes as fast as possible. That is, it is not synchronized to real time. This behavior may be desirable if you are running code on the debugger or hardware simulator.

---

**Caution**   When using device driver blocks from the Embedded Target for Infineon C166 Microcontrollers libraries 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 operates incorrectly. See the C166 Resource Configuration reference page for further information.

---

**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.



DO NOT
COPY

C166
Configuration
Class

---

**Caution**   Configuration class blocks exist only to provide information to other blocks. *Do not copy these objects into a model.* If you do you see an error dialog box to warn you. This causes build failures.

---

# Overview of C166 Configuration Parameters

In the **C166 Options** (under **Real-Time Workshop** in the Configuration Parameters dialog) some C166-specific options are available:

**Include input/output driver function hooks**
> Use this option to integrate your own device driver code. This is described in "Calling the Device Driver Functions from c166_main.c" on page 3-6.

**Maximum number of concurrent base-rate overruns**
> Option for task execution profiling. See "Overview of Execution Profiling" on page 5-2.

**Maximum number of concurrent sub-rate overruns**
> Option for task execution profiling. See "Overview of Execution Profiling" on page 5-2.

**Execution profiling**
> Option for task execution profiling. See "Overview of Execution Profiling" on page 5-2.

**Number of data points**
> Option for task execution profiling. See "Overview of Execution Profiling" on page 5-2.

# Tutorial: Simple Example Applications for C166 Microcontrollers

This section includes the following topics:

# 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 and c166_serial_io

- The Tasking C Cross-Compiler and Tasking CrossView Pro Debugger for compiling and downloading generated code to the target hardware

  As an alternative to CrossView, you can use the MiniMon utility for downloading an application to your target hardware.

Using these, you can build the complete application. 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 Chapter 6, "Blocks — By Category".

# Tutorial: Creating a New Application

In this tutorial, you build stand-alone real-time applications from models incorporating blocks from the Embedded Target for Infineon C166 Microcontrollers library.

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 CrossView Pro Debugger to run a model on the C166 Simulator or debug on-chip

## Before You Begin

We assume that you are already familiar with Simulink and with the Real-Time Workshop code generation and build process. 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 C Cross-Compiler and CrossView Pro Debugger
- MiniMon download utility

You must make sure the target preferences have been set correctly. See "Setting Target Preferences" on page 1-17.

---

**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 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.
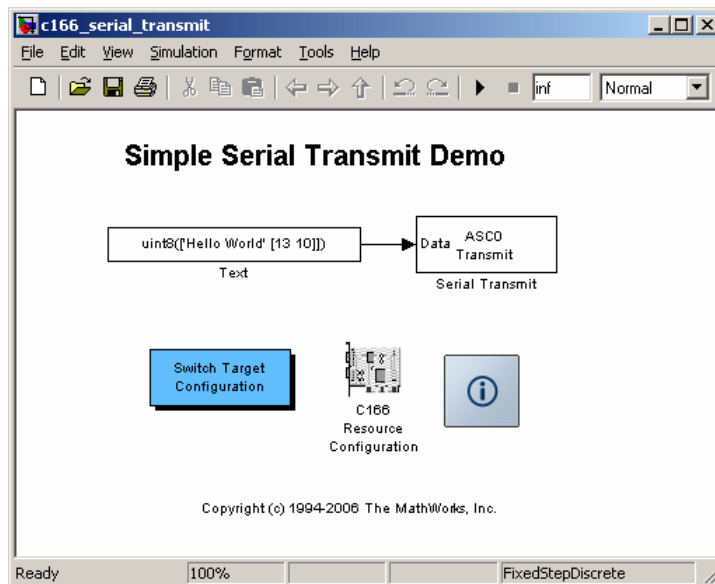


The model contains a C166 Resource Configuration object. When building a model with driver blocks from the Embedded Target for Infineon C166 Microcontrollers library, you must always place a C166 Resource Configuration object into the model (or the subsystem from which you want to generate code) first.

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 ASC0 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 these clock busses will run.

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

**2** Double click the switch target configuration block, and then select c167cs_hw. This selection sets the appropriate System_frequency and External_oscillator_frequency in the Resource Configuration block and the Link for TASKING option set.

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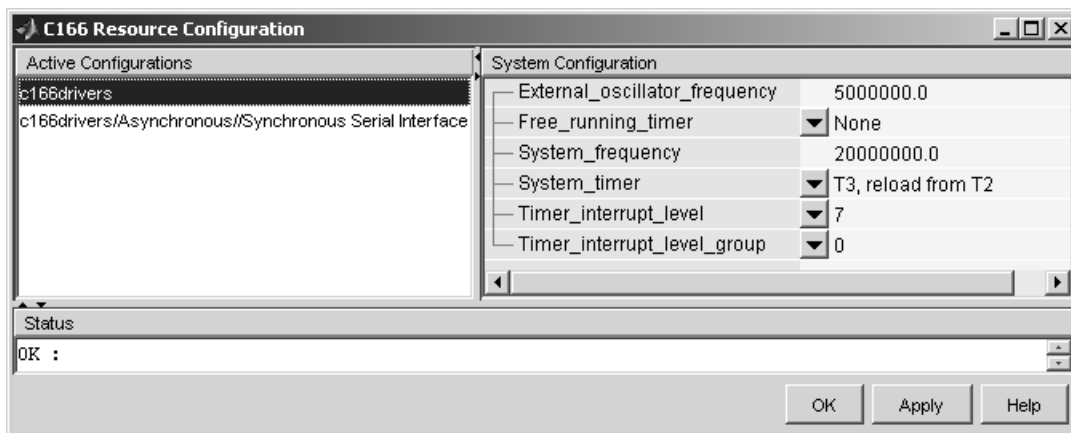


In this tutorial, use the default C166 Resource Configuration settings.

**Note** If hardware is running at a system frequency other than 20 MHz, 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.

**4** View the settings for the serial driver block by clicking the c166drivers/Asynchronous/Synchronous Serial Interface option in the **Active Configurations** pane. These settings are shown in the following illustration.
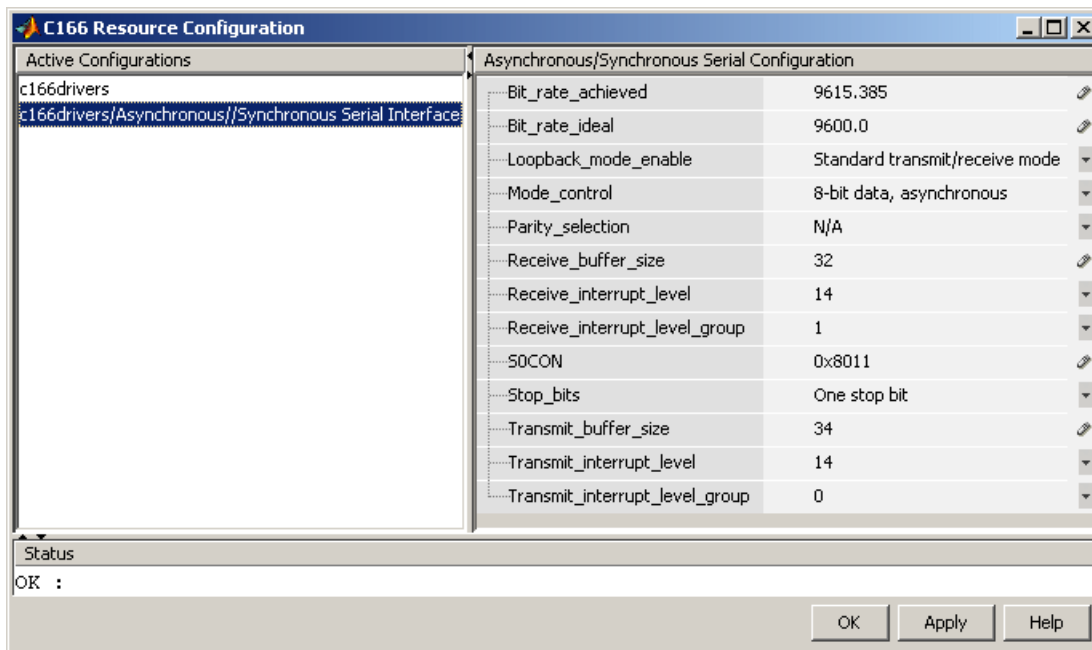


The settings appear in the **Asynchronous/Synchronous Serial Configuration** pane on the right. Do not edit any of these parameters for this tutorial. To learn more about the C166 Resource Configuration object, see C166 Resource Configuration.

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

## Generating and Downloading Code

To generate code for the model:

**1** Select **Simulation > Configuration Parameters**.

The Configuration Parameters dialog opens.

**2** Select **Real-Time Workshop** in the tree, as shown below.
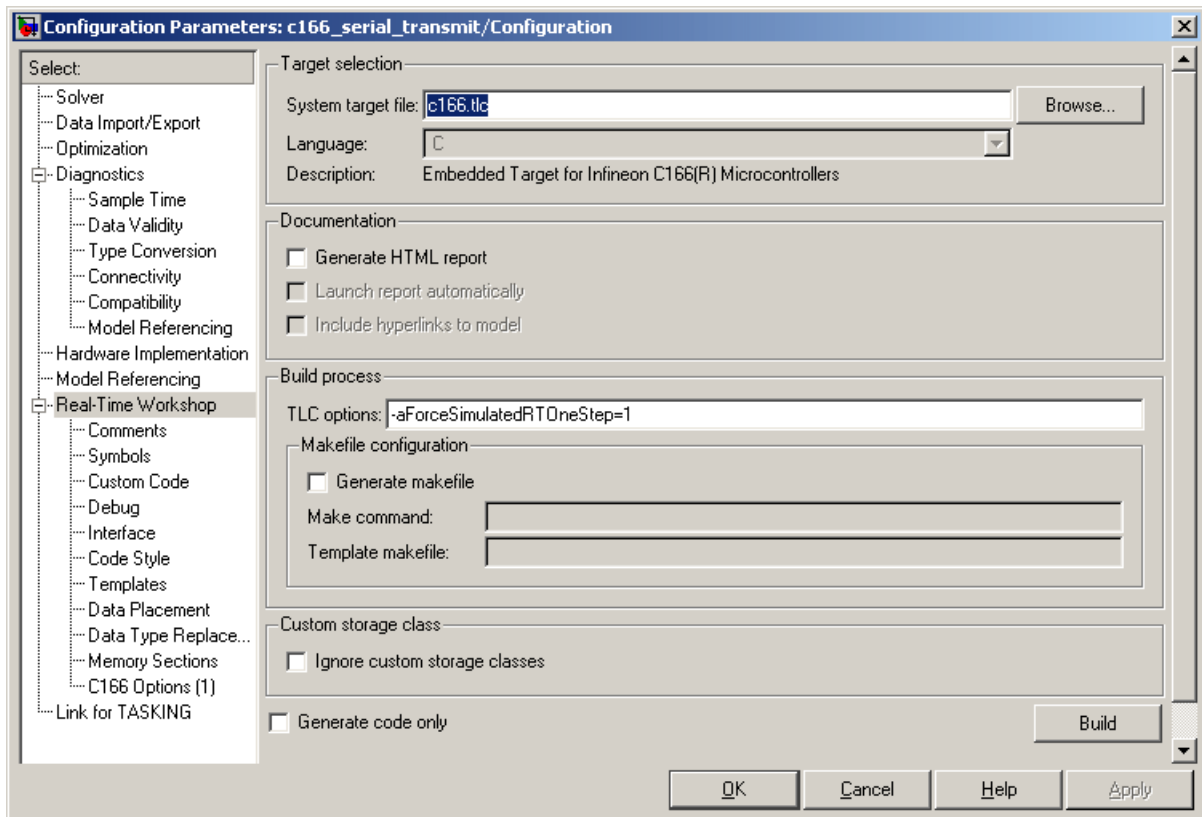


**3** Click **Build**.

Alternately, you can go straight to building the model by selecting **Tools > Real-Time Workshop > Build Model** or using the shortcut **Ctrl+B**.

Watch the progress messages in the command window as code is generated.

**4** Click the MiniMon link in the MATLAB command window to download your application via MiniMon. If the Minimon link has not been generated then your C166 Link for TASKING option set is not compatible with MiniMon downloads. This failure to generate could be because you are targeting one of the simulator configurations or your board is using OCDS (on-board wiggler) to connect to the target. The Minimon option should appear when:

- The Link for TASKING Build Action is set to **Create and Build Application Project**, **Create, Build and Execute Application Project**, or **Create, Build and Debug Application Project**.

- The option set is for hardware (rather than simulator).

- You are using a serial connection to connect to your target.

- If you have created your own template projects, the option to generate a hex file must be selected.

---

**Caution** You must ensure the option to generate a hex file is turned on. If you do not you will see the following warning:

```
It was not possible to generate a minimon script for this
 build. This is because your EDE project template is not
configured to generate a .hex file which is required by
Minimon. To generate a .hex file as part of the build
you need to check the box 'Intel HEX records' in your
EDE project template.
You can change this option via Project -> Project Options
 -> Linker/Locator -> Output Format.
```

---

When MiniMon is started, a dialog box appears asking you to reset your hardware.

**5** 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 then disappears and the code begins 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 and a linefeed over the serial interface.

## Example 2: c166_serial_io

This example model 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.
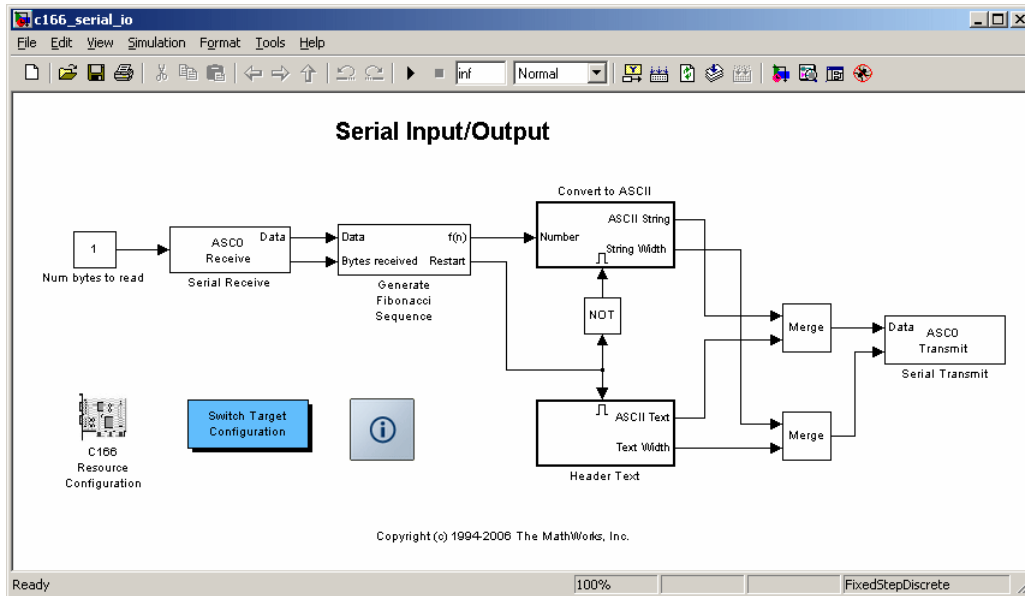
**2** Double click on the switch target configuration block, then select `c167cs_hw`. This will set the `System_frequency` and `External_oscillator_frequency` in the Resource Configuration block and the Link for TASKING option set.

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

Watch the progress messages as code is generated from the model.

**4** You can download the application by clicking on the link at the end of the build log. This link launches Minimon.

MiniMon is started to download the code to the target over the serial connection. The MiniMon dialog appears asking you to reset your hardware.

**5** 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 then disappears and the code begins 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 restarts at the beginning of the sequence. Examine the model to see how this works: the Serial Receive block passes the restart command through 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 use another demo model, `c166_user_io`. 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 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 conflicts 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 or JTAG is available.

**1** Open the model `c166_user_io`.

**2** Select **Simulation > Configuration Parameters**.

**3** Select **Link for TASKING** in the tree.
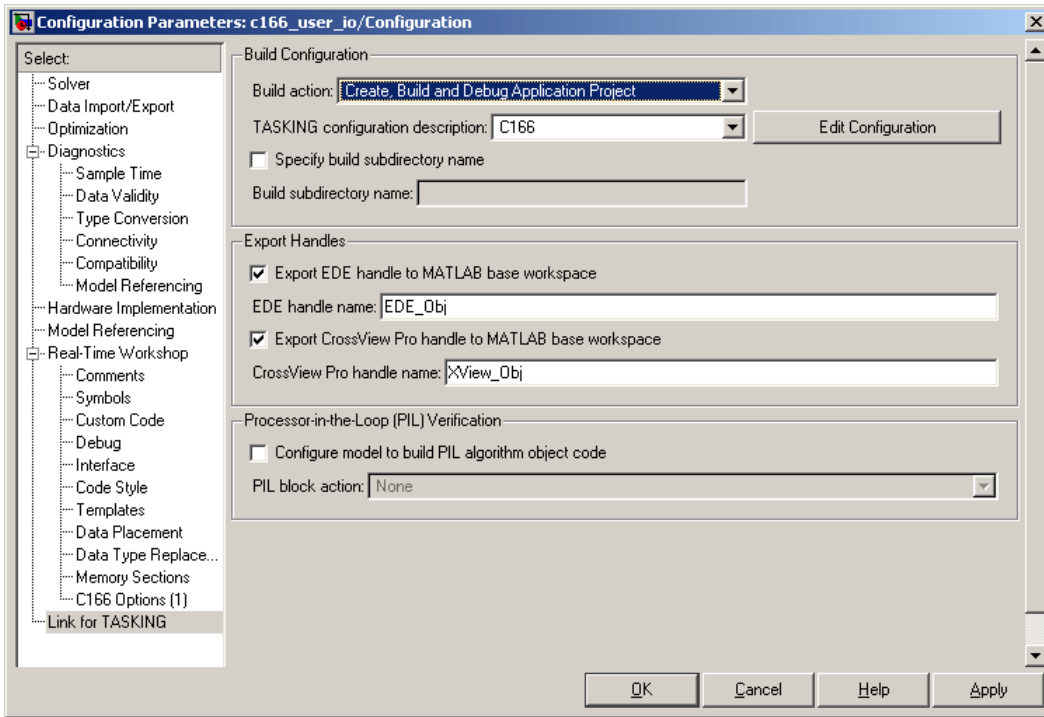
4 Select the **Build action** `Create, Build and Debug Application Project`.

5 Before generating code, check that your target preferences related to the debugger are correctly configured. See "Setting Target Preferences" on page 1-17.

6 Click **OK**.

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

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

Watch the progress messages in the command window as code is generated. At the end of the build process, your debugger launches 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. If you attempt to use the debugger once your application is running, you will no longer be able to control the application from the debugger, because the application is using the serial channel.

## Fixed-Point Example Model: c166_fuelsys

The c166_fuelsys model is derived from the demo c166_fuelsys. The floating point control algorithm from the original model has been converted to fixed point to allow efficient code generation for the Infineon C166 microcontroller.

> **Note** This demo requires Simulink Fixed Point.

The complete model 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 subsystem only:

**1** Open the model c166_fuelsys.
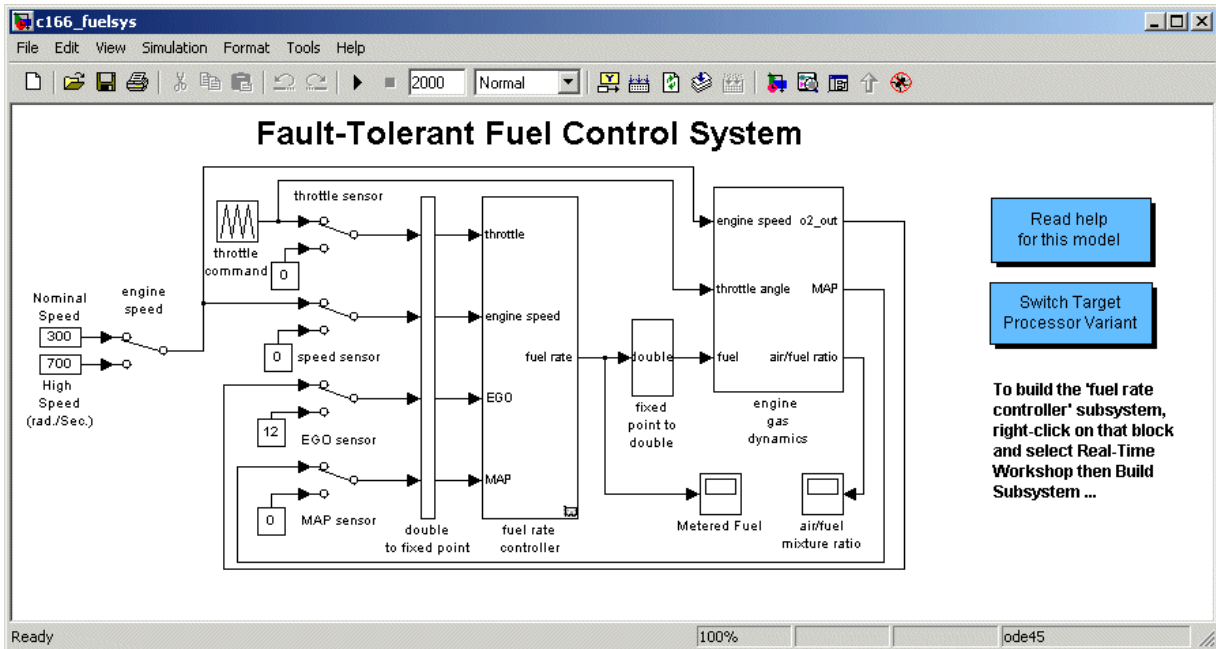
**2** Select **Simulation > Configuration Parameters**.

**3** Select Real Time Workshop in the tree, and then select the check box **Generate HTML report**.

**4** Make sure that the **Generate code only** option is not selected. The reason for this step is that the code generation report obtains information from MAP files that are created by your cross-compiler during the build process. If the **Generate code only** option is on, these files are not generated, which prevents the generation of the code generation report.

**5** Select Link for TASKING in the tree, and observe the **Build Action** is **Create and Build Application Project**. You must have one of the Build options selected to get the code profile report (with RAM/ROM usage):

  • **Create and Build Application Project**

  • **Create, Build and Execute Application Project**

  • **Create, Build and Debug Application Project**

**6** Close the Configuration Parameters dialog box.

**7** Right-click the fuel rate controller block.

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

**9** 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 the link `Code profile report` in the left list. If you compared with the original floating-point version of the `fuelsys` control algorithm: you would find that using the fixed-point implementation results in a considerable reduction in both RAM and ROM.

# Generating ASAP2 Files

ASAP2 is a data definition standard 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. See also "Compatibility with Calibration Packages" on page 7-26.

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 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 > Configuration Parameters**.

The Configuration Parameters dialog appears.

**2** Select **Interface** (under **Real-Time Workshop)** in the tree.

**3** Select the ASAP2  option from the **Interface** drop-down menu, in the **Data exchange** frame, as shown following.
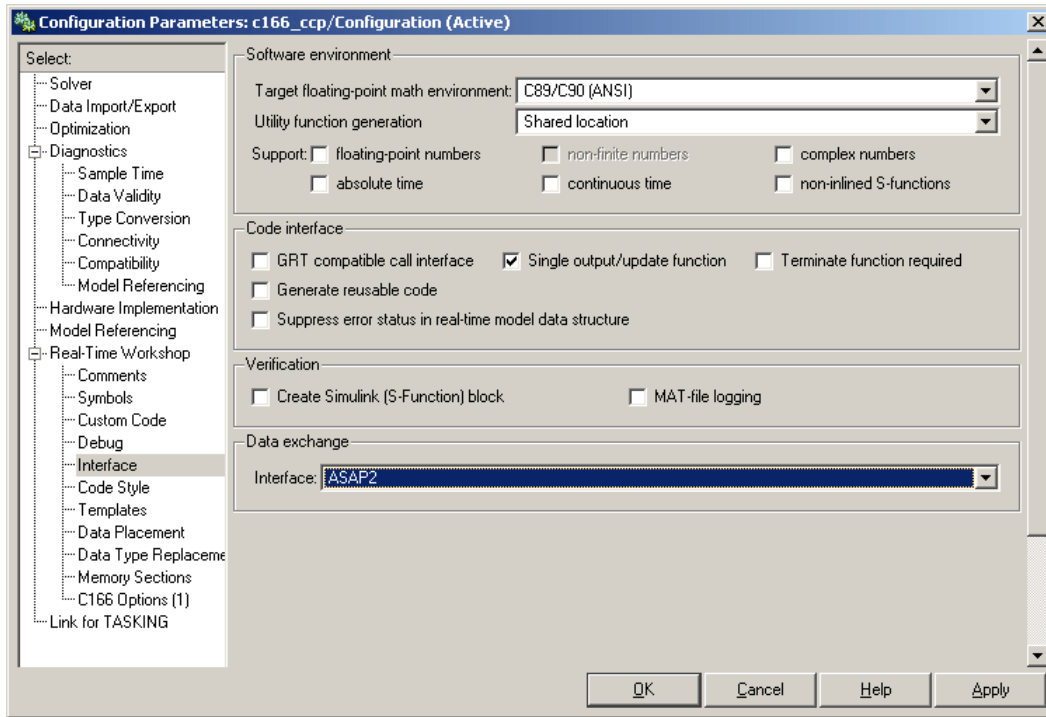
**4** Click **Apply**.

The build process creates an ASAM-compliant ASAP2 data definition file for the generated C code.

- The 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 postprocessing the generated file.

- The map file options in the template project need to be set up a certain way for this procedure to work. If you have created your own template projects, and you do not have the correct settings, you see the following instructions:

```
Warning: It was not possible to do ASAP2 processing on your
.map file.This is because your EDE project template is not
configured to generate a .map file in the correct format.
```

```
To generate a .map file in the correct format you need to
setup the following options in your EDE project template:
Generate section map should be checked on
Generate register map should be checked off
Generate symbol table should be checked on
Format list file into pages should be checked off
Generate summary should be checked off
Page width should be equal to 132 characters
Symbol colums should be 1
You can change these options via Project -> Project Options
 -> Linker/Locator -> Map File -> Map File Format.
```

Embedded Target for Infineon C166 Microcontrollers performs this postprocessing 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 postprocessing is performed automatically and requires no additional input from you.

For an example of a model that is configured to generate an ASAP2 file, see c166_ccp.

# 3

# 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` 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 tell Real-Time Workshop 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`
demo model.

For a guide to creating device drivers, see "Developing Device Drivers for
Embedded Targets" in the Developing Embedded Targets for Real-Time
Workshop Embedded Coder documentation.

# Integrating Hand-Coded Device Drivers with a Simulink Model

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

This approach 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`.

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 need to reconfigure the I/O behavior frequently. If you want to take this alternative approach, you should consult the documentation on S-functions and TLC. See the section Developing Device Drivers for Embedded Targets in the document Developing Embedded Targets for Real-Time Workshop Embedded Coder.

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

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

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

# Preparing Input and Output Signals to the Device Driver Functions

Structure your model similarly to c166_user_io. Place the control algorithm that will be targeted onto the C166 microcontroller hardware in a separate subsystem. Before 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 chooses 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.

**2** Open the controller subsystem.

**3** Click the output_PWM0 signal.

**4** Select the menu item **Edit > Signal Properties**.

The **Signal Properties** dialog appears, as in the example following.
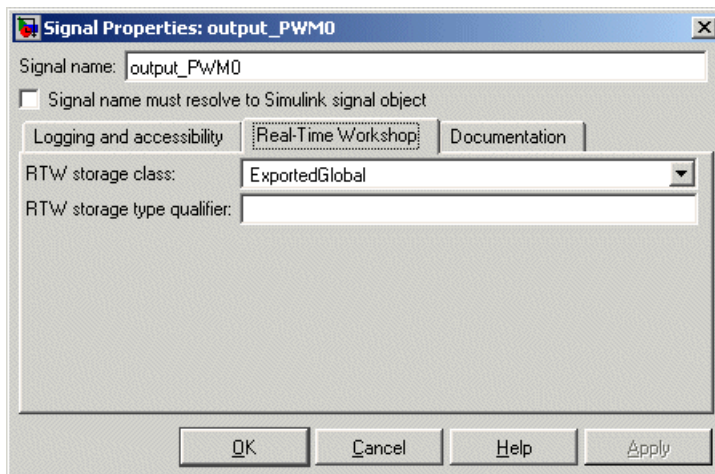
**5** Enter the required variable name for your signal in the **Signal name** edit box. This must match the variable name required by your hand written device driver functions.

**6** Click the **Real-Time Workshop** tab and select `ExportedGlobal` from the **RTW storage class** drop-down menu.

When you generate code for this model, Real-Time Workshop uses the variable name that you have specified and creates 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` 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 > Configuration Parameters**.

The Configuration Parameters dialog appears.

**2** Select **C166 Options (1)**, under **Real-Time Workshop** in the tree, as shown in the example below.
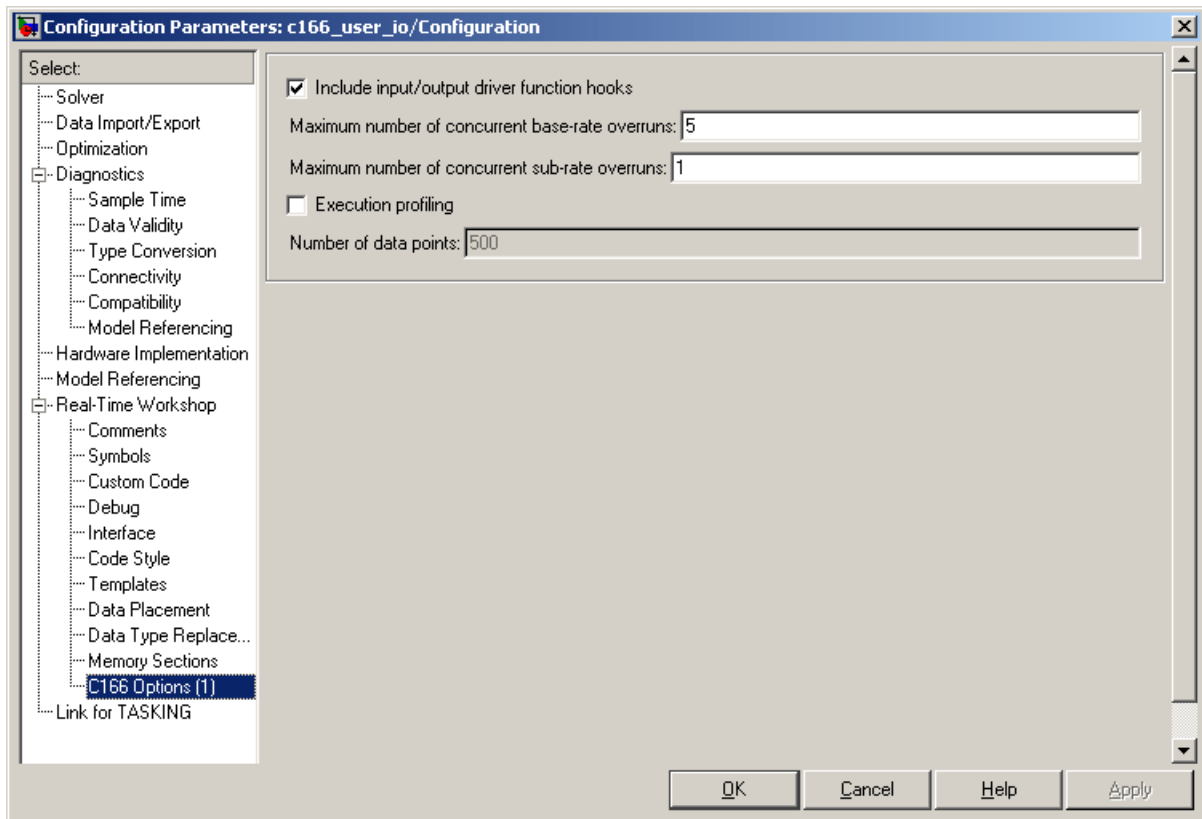
**3** Select the check box option for including I/O driver function hooks.

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 sub-rate 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 Workshop build process to compile and link the I/O driver source files that you have written. You do so by adding the files to the custom code dialog box. Access the Configuration Parameters dialog box, look under Real-Time Workshop > Custom Code, and add the necessary Include Directories and Source Files, as shown in the following figure.
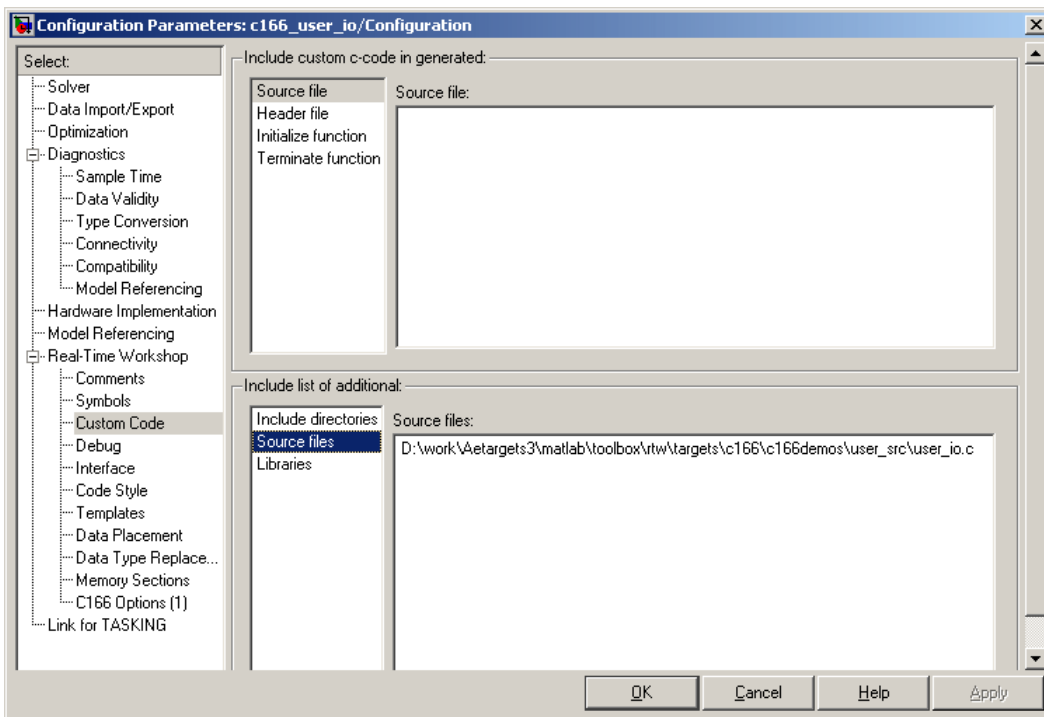


You are now ready to build your model and run it in real time.

You can examine an example of this 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 generate code from the `controller` subsystem, which automatically downloads and runs 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`.
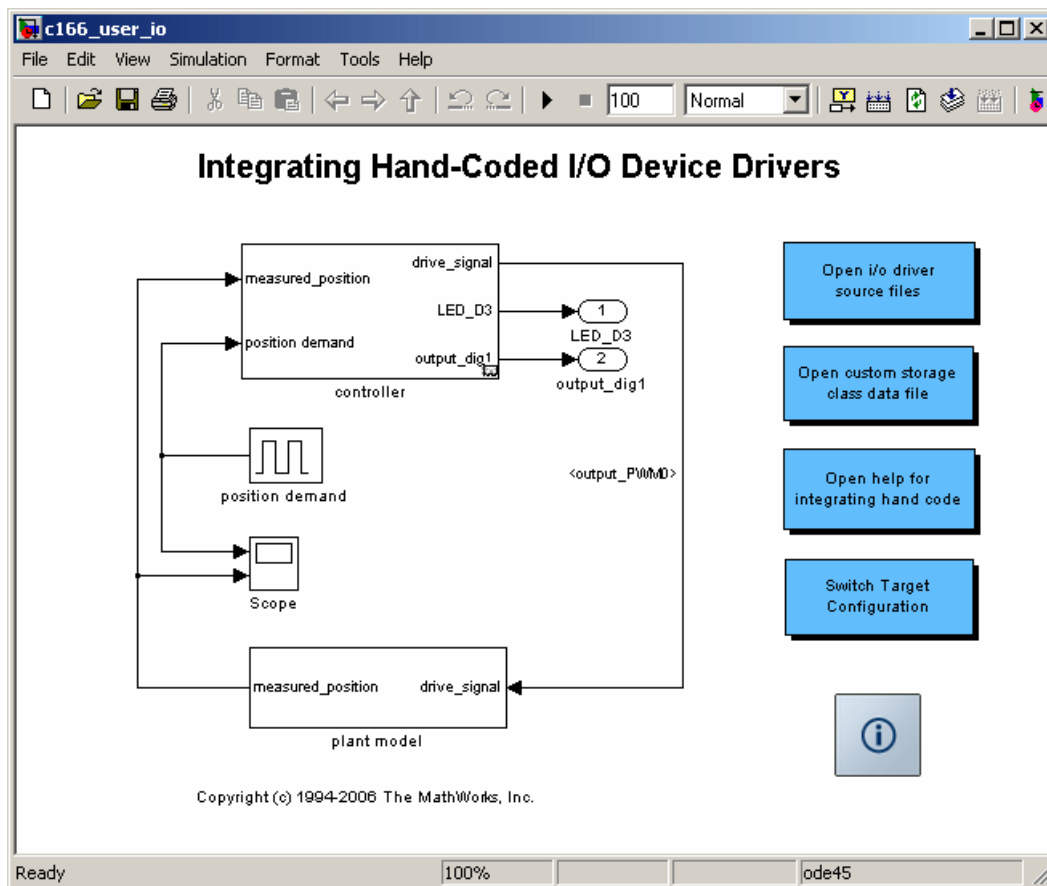
**2** Open the `controller` subsystem by double-clicking and select the signal `input_adc0`.
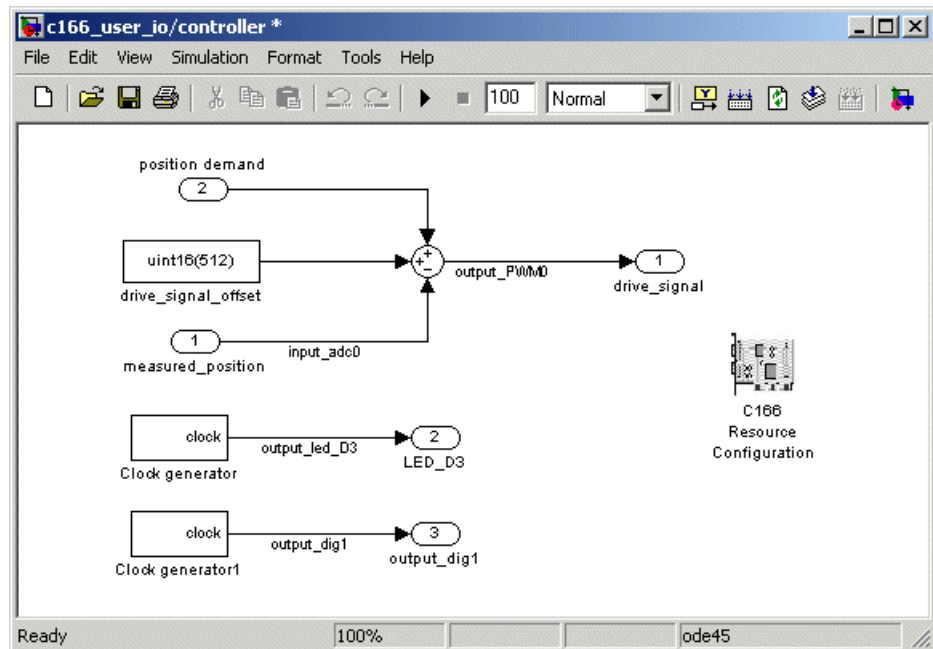
**3** Select the menu item **Edit > Signal Properties**.
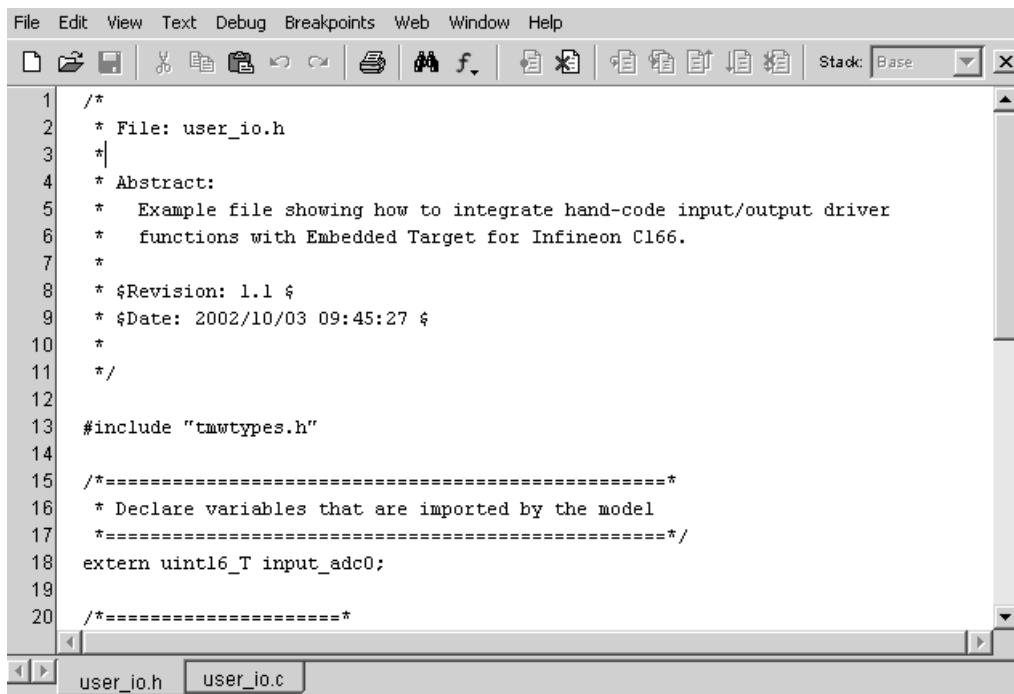
The Signal Properties dialog appears.
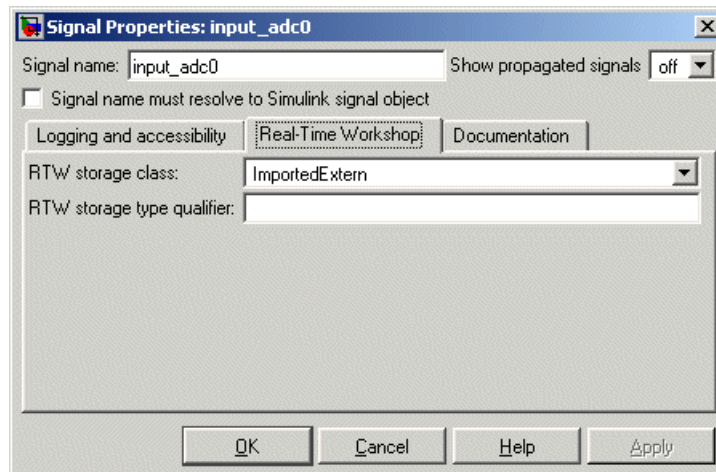
Click the **Real-Time Workshop** tab and observe that the **RTW storage class** is ImportedExtern. When you generate code for this model, Real-Time Workshop uses the specified variable name input_adc0 and creates an extern declaration in the model header file. Since the Real-Time Workshop storage class 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** In the top level model, double-click the link **Open the i/o driver source files**.

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

```
File  Edit  View  Text  Debug  Breakpoints  Web  Window  Help

 1    /*
 2     * File: user_io.h
 3     *
 4     * Abstract:
 5     *    Example file showing how to integrate hand-code input/output driver
 6     *    functions with Embedded Target for Infineon C166.
 7     *
 8     * $Revision: 1.1 $
 9     * $Date: 2002/10/03 09:45:27 $
10     *
11     */
12
13    #include "tmwtypes.h"
14
15    /*=================================================*
16     * Declare variables that are imported by the model
17     *=================================================*/
18    extern uint16_T input_adc0;
19
20    /*===================*

     user_io.h    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** You need to instruct Real-Time Workshop to compile and link the hand-coded I/O driver source files in the build process. You do so by adding the files to the custom code dialog box. Access the Configuration Parameters dialog box, select Real-Time Workshop > Custom Code in the tree, and review the Include Directories and Source Files, as shown in the following figure.
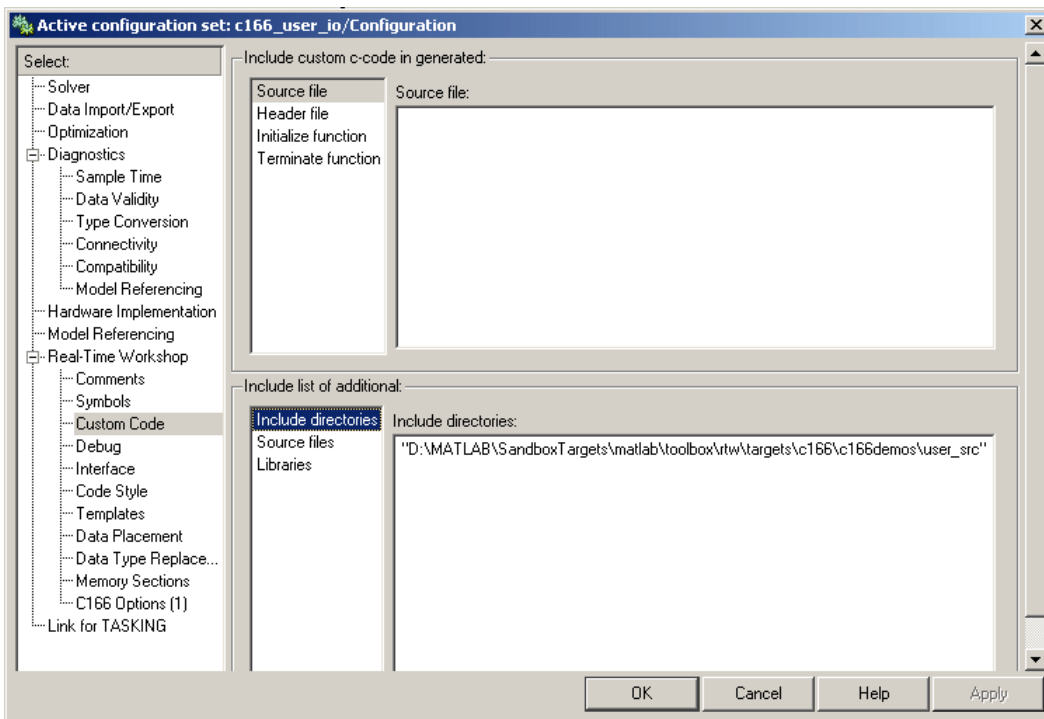


**7** Select **C166 Options (1)** (under **Real-Time Workshop** in the tree). Observe the selected option **Include input/output driver function hooks**.
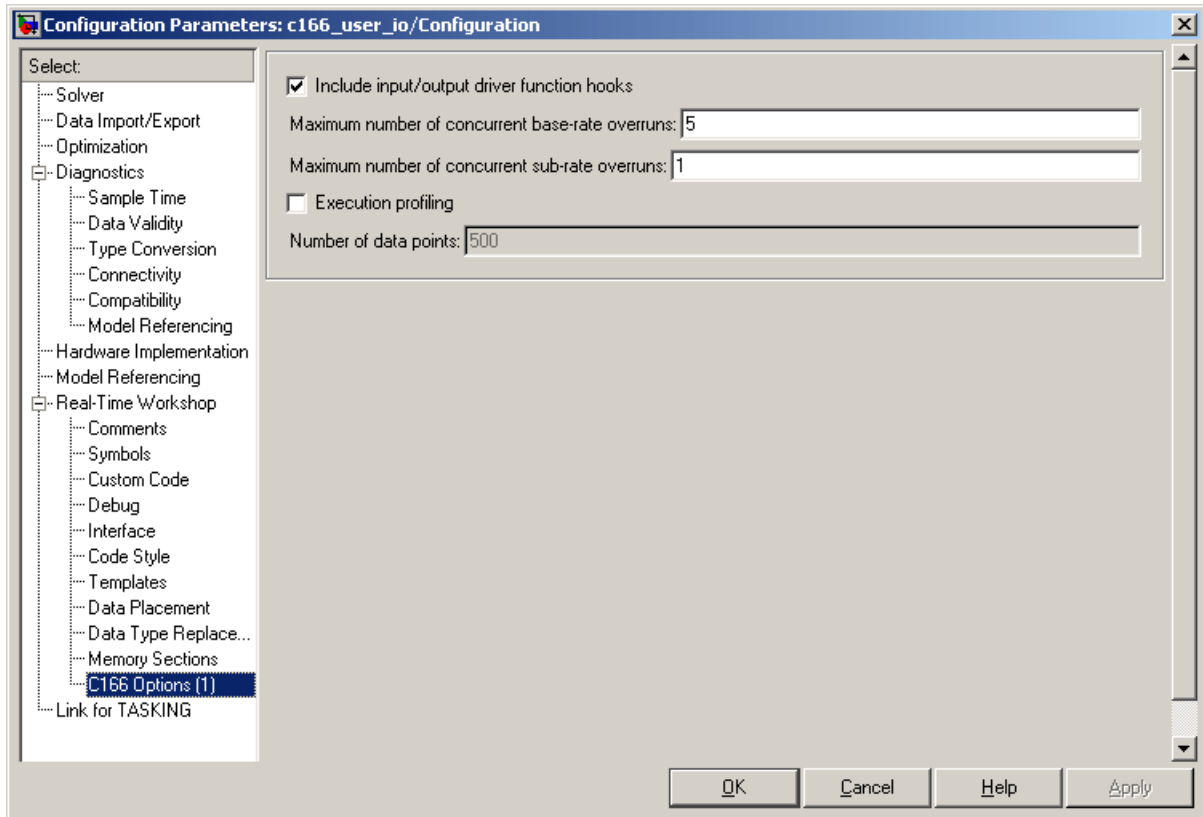
This instructs Real-Time Workshop to include extra calls to the user-supplied I/O device driver functions when code is generated for this model.

**8** Select **Interface** in the tree. Observe the option **Floating-point numbers** is *not* selected.

If your model does not use floating point, you should make sure this option is not checked to use integer code only. Using only integer code results in smaller code size and faster real-time execution. It also speeds up the build process because libraries that are used only 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).

**9** Close the Signal Properties dialog and Configuration Parameters dialog if they are still open.

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.

**10** Generate code by right-clicking the controller subsystem and selecting **Real-Time Workshop > Build Subsystem**.

**11** Click **Build** in the Build code for Subsystem: Controller dialog that appears. Watch the messages as the process proceeds and code is generated.

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.

# 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`. Included is a comparison with another custom storage class variable in `c166_user_io`

# 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 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 data type 'boolean'.

**2** Attach a label to the signal, either by using **Edit > Signal Properties** or by double-clicking 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** Select **View > Model Explorer** and click the base workspace to inspect all the Simulink data objects that are available to the model.

**5** Build the model.

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. 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` to see the automatic debugger start at the end of the build.

Follow these steps:
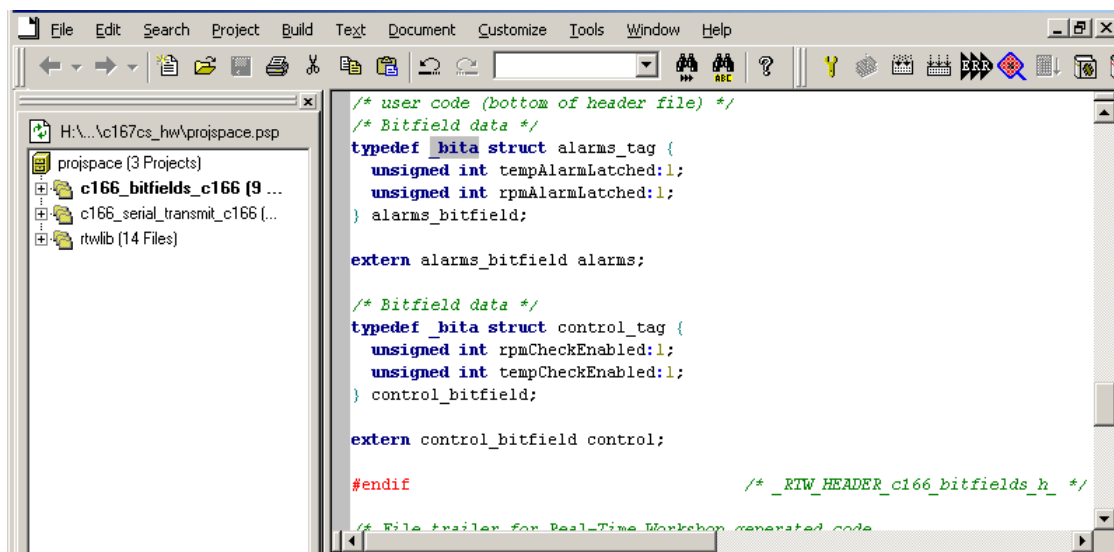
**1** Open `c166_bitfields`.
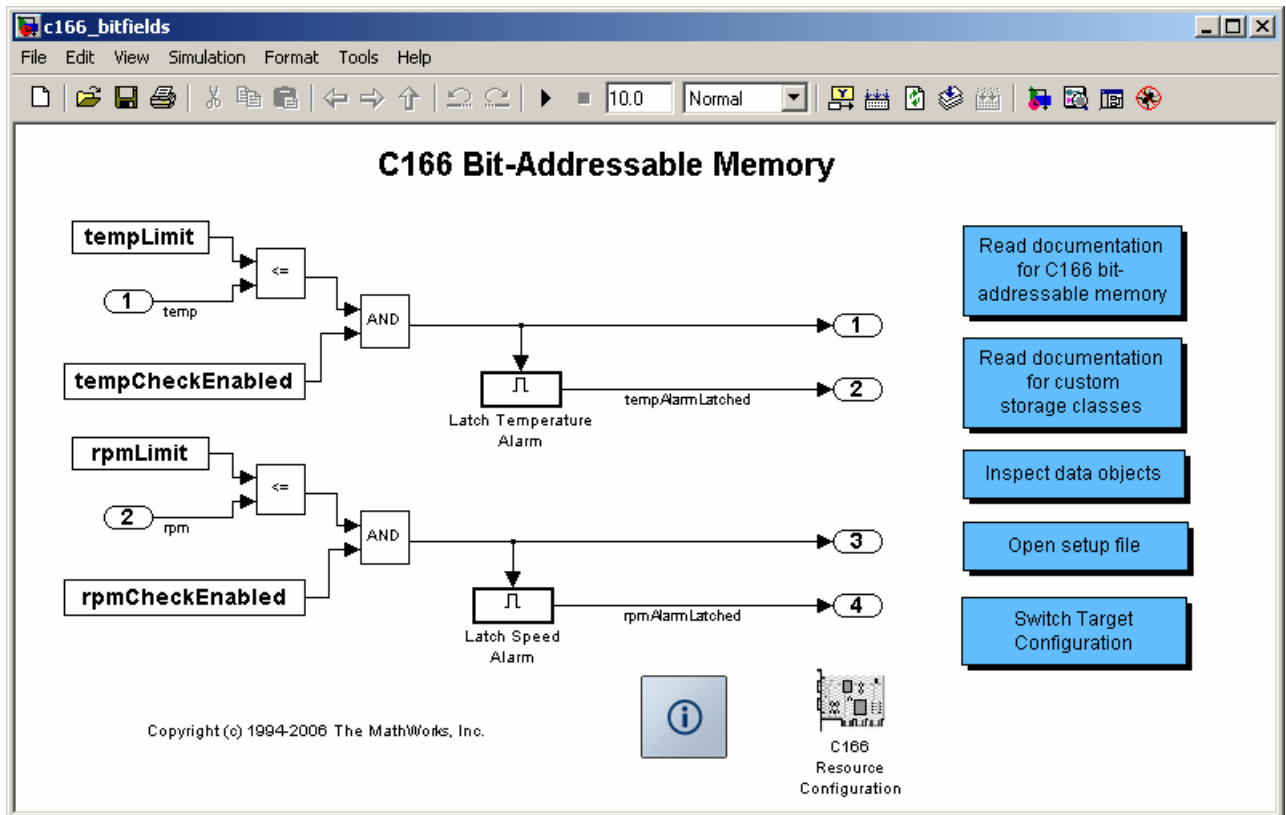


**2** Press **Ctrl+B** to build the model.

**3** Examine the project generated code in the TASKING EDE:

**a** Select **Search > Multiple Sources**.

**b** In the dialog box, select Project Space under Multiple Sources, and enter `_bita` for the search string.



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

```
File  Edit  View  Text  Debug  Breakpoints  Web  Window  Help

 1  │ % C166BITFIELDDATA create data for C166 bitfield demo model
 2  │
 3  │ %    Copyright 2002 The MathWorks, Inc.
 4  │ %    $Revision: 1.1 $
 5  │ %    $Date: 2002/10/03 09:45:24 $
 6  │
 7  │-  cscdemoclearws
 8  │
 9  │-  tempAlarm = SimulinkC166.Signal;
10  │-  tempAlarm.RTWInfo.CustomAttributes.BitFieldName = 'alarms';
11  │
12  │-  tempAlarmLatched = SimulinkC166.Signal;
13  │-  tempAlarmLatched.RTWInfo.CustomAttributes.BitFieldName = 'alarms';
14  │
15  │-  rpmAlarm = SimulinkC166.Signal;
16  │-  rpmAlarm.RTWInfo.CustomAttributes.BitFieldName = 'alarms';
17  │
18  │-  rpmAlarmLatched = SimulinkC166.Signal;
19  │-  rpmAlarmLatched.RTWInfo.CustomAttributes.BitFieldName = 'alarms';
20  │
21  │-  tempLimit = uint16(500);
```

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 documentation for custom storage classes** in
the model to go directly to the relevant Real-Time Workshop Embedded
Coder help section.

**5** You can double-click **Inspect data objects** to inspect all the Simulink data
objects that are available to the model.
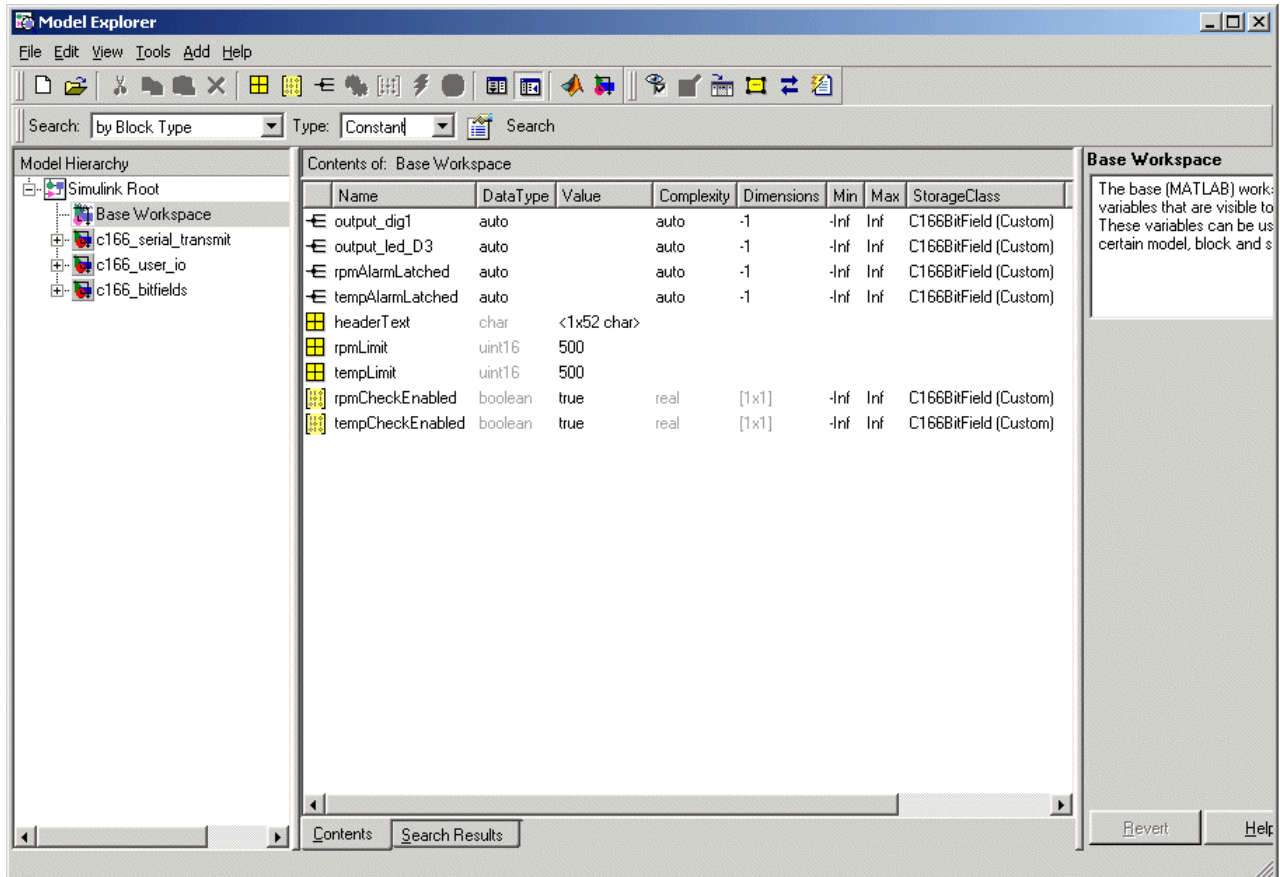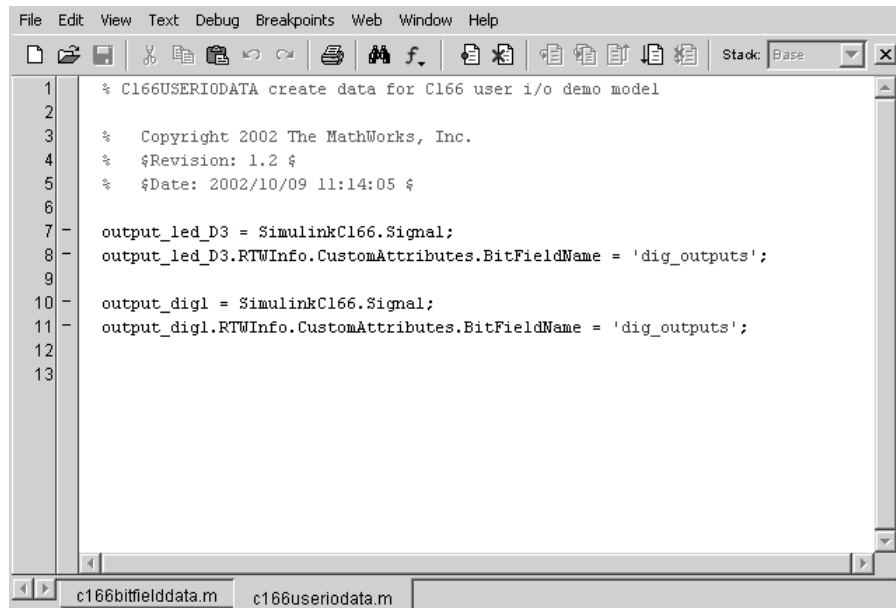
Here you can see the `SimulinkC166.Signal` data object and you can click on each object to inspect the properties.

**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`.

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

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

```
File  Edit  View  Text  Debug  Breakpoints  Web  Window  Help

 1      % C166USERIODATA create data for C166 user i/o demo model
 2
 3      %   Copyright 2002 The MathWorks, Inc.
 4      %   $Revision: 1.2 $
 5      %   $Date: 2002/10/09 11:14:05 $
 6
 7      output_led_D3 = SimulinkC166.Signal;
 8      output_led_D3.RTWInfo.CustomAttributes.BitFieldName = 'dig_outputs';
 9
10      output_dig1 = SimulinkC166.Signal;
11      output_dig1.RTWInfo.CustomAttributes.BitFieldName = 'dig_outputs';
12
13

   c166bitfielddata.m    c166useriodata.m
```

Compare with `c166bitfielddata.m`.

For more details on the variables in this model, see "Tutorial: Using the Example Driver Functions" on page 3-9.

# Execution Profiling

This section contains the following topics:

# Overview of Execution Profiling

Embedded Target for Infineon C166 Microcontrollers provides a set of utilities for recording, uploading, and analyzing execution profile data for timer-based tasks and asynchronous Interrupt Service Routines (ISRs). With these utilities, you can

- Generate a graphical display that shows when timer-based tasks and interrupt service routines are activated, preempted, resumed, and completed.

- Generate a report with information on

  - Maximum number of overruns for each timer-based task since model execution began

  - Maximum turnaround time for each timer-based task since model execution began

  - Analysis of profiling data for timer-based tasks and asynchronous interrupts over a period of time

To perform execution-profiling analysis on a model, you must perform the following steps:

**1** Place a copy of the appropriate execution profiling block in your model:

  - Execution Profiling via ASC0 if using a serial connection

  - Execution Profiling via CAN A if using CAN with a C166 processor

  - Execution Profiling via TwinCAN A if using CAN with an XC16x processor variant

**2** Select the **Execution profiling** option under Real-Time Workshop options in the Configuration Parameters dialog. See "Real-Time Workshop Options for Execution Profiling" on page 5-5.

**3** Connect the target processor to your host PC (with a serial or CAN cable).

**4** Build, download, and run the model.

**5** Initiate execution profiling by running the command `profile_c166`. See below for more information on the profiling command.

Two forms of execution profiling are provided:

**1** The worst-case values for task turnaround times and number of concurrent task overruns since model execution began are updated whenever a previous worst-case value is exceeded.

**2** A snapshot of task and ISR activity may be recorded over a period of time; the length of this period depends on how much memory is reserved to log the data.

## The Profiling Command

Use the profiling command as follows:

```
profile_c166(connection)
```

Specify your `connection` as `'can'` or `'serial'`, for use with the CAN or ASC0 execution profiling blocks.

Optionally, you can specify your configuration as follows:

```
profdata = profile_c166(connection, 'ModelName', modelname)
```

This specification automatically sets the bit rate by analyzing *modelname* and extracting the correct connection bit rate setting from the model. *modelname* should be set to the name of a model which is currently open and running on the target.

The execution profiling data is returned in the format documented by `exprofile_unpack`.

Alternatively, you can set the bitrate for the connection to the target manually as follows:

```
profdata = profile_c166(connection, 'BitRate', bitrate)
```

`bitrate` must be the same as the bit rate specified for the application that is running on the target.

## Definitions

**Task turnaround time** is the elapsed time between start and finish of a task. If the task is not preempted, then the task turnaround time is equal to the task execution time.

**Task execution time** is that part of the time between task start and finish when the task is actually running and not preempted by another task. Note that the task execution time cannot be measured directly, but is inferred from the task start and finish time and the intervening periods during which it was preempted by another task. Note that, in performing these calculations, no account is taken of processor time consumed by the scheduler while switching tasks: this means that, in cases where preemption has occurred, the reported task execution times will overestimate the true values.

**Concurrent task overruns** occur when a timer task does not complete before that same task is next scheduled to run. Depending on how the real-time scheduler is configured, a task overrun may be handled as a real-time failure. Alternatively, a small number of concurrent task overruns may be allowed to accommodate cases where a task occasionally takes longer than normal to complete.

## Execution Profiling Blocks

See the block reference sections:

- C166 Execution Profiling via ASC0
- C166 Execution Profiling via CAN A
- C166 Execution Profiling via TwinCAN A

# Real-Time Workshop Options for Execution Profiling

You can see these options by selecting **C166 Options (1)** (under **Real-Time Workshop** in the tree) in the Configuration Parameters dialog.
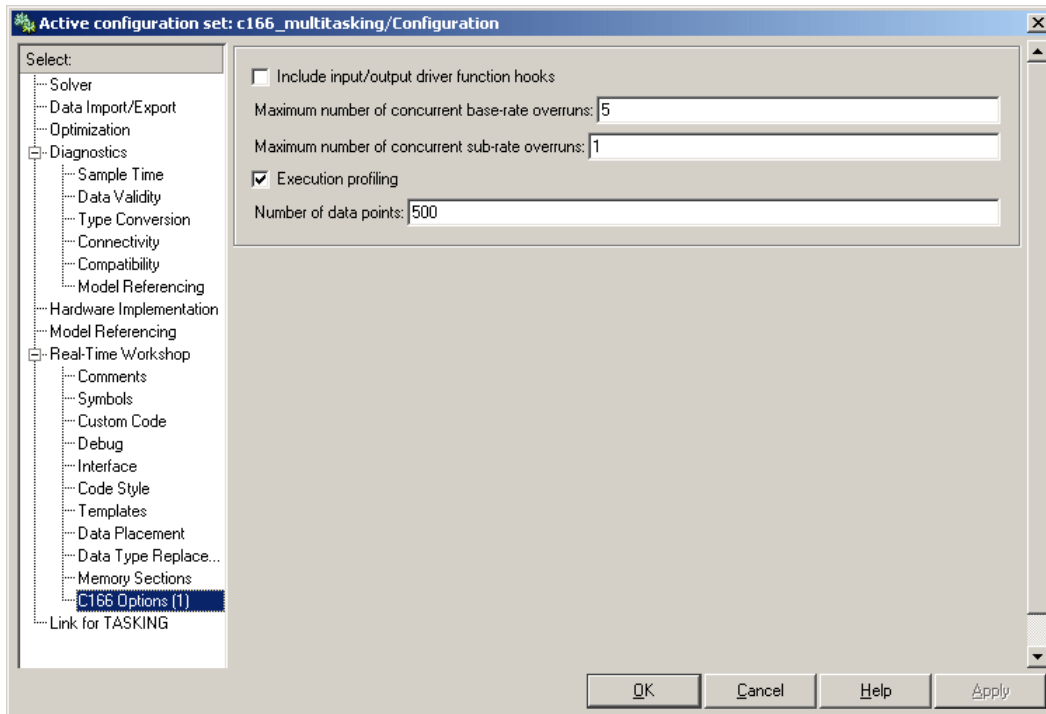


## Execution Profiling

If this option is selected, then the generated code for the model will be "instrumented" with function calls at the beginning and end of each task or ISR to be profiled. These function calls read a timer (on C166 a free running timer is selected from the options in the C166 Resource Configuration block) and log this reading along with a task identifier.

When code for the model is generated, these functions will update data on the worst-case turnaround time for each timer-based task as well as the worst-case number of concurrent task overruns, whenever a previous

worst-case value is exceeded. Additionally, when a trigger is provided, data will be logged over a period of time to record all task start and finish times. The trigger signal can be supplied, for example, by the block C166 Execution Profiling via CAN A.

## Number of Data Points

When a snapshot of task and ISR activity is logged, this data is stored in memory that is statically allocated at build time. Each data point requires 4 bytes on C166. The larger the number of data points to be stored, the more RAM that must be reserved for this purpose. At the end of a logging run, the data must be uploaded to the host computer for analysis; this is typically achieved by using one of the C166 execution profiling blocks — via ASCO, CAN A, or TwinCAN A. See the reference pages for C166 Execution Profiling via ASC0, C166 Execution Profiling via CAN A, and C166 Execution Profiling via TwinCAN A.

## Task Scheduler Overrun Options

These scheduler options configure the allowable number of concurrent task overruns. You can see these options on the **C166 Options (1)** section in the Configuration Parameters dialog.
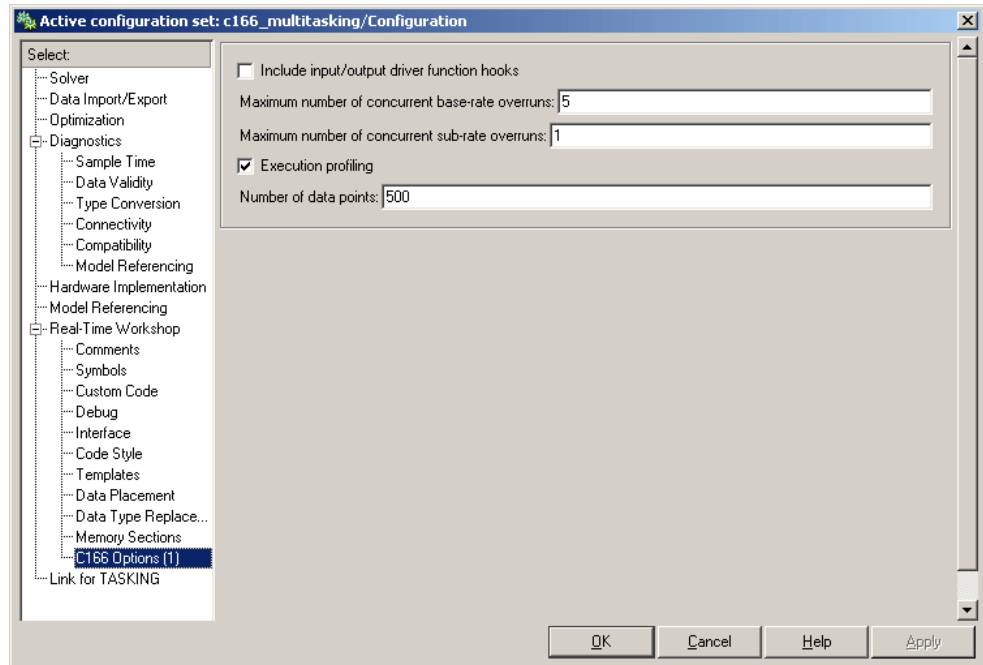
You can use the options **Maximum number of concurrent base-rate overruns** and **Maximum number of concurrent sub-rate overruns** to configure the behavior of the scheduler when any of the timer based tasks do not complete within their allowed sample time. It is useful to allow task overruns in the case where a task may occasionally take longer than usual to complete (e.g., if extra processing is required when a special event occurs); if the task overrun is only occasional, then it is possible for the scheduler to catch up after the extra processing has been completed.

If the maximum number of concurrent overruns for any task is exceeded, this is deemed to be a failure and the real-time application is stopped.

As an example, if the base rate is 1 ms and the maximum number of concurrent base-rate overruns is set to 5 then it is possible for the base rate task to run for almost 6 ms before failure occurs. Once the overrun has occurred, it is necessary for subsequent executions of the base rate to complete in less than 1 ms in order that the lost time is recovered.

The occurrence of base-rate overruns does not affect the numerical behavior of the algorithm (although reading/writing external devices will of course be delayed).

If sub-rate overruns are allowed, then the transfer of data between different rates (via rate-transition blocks) in the model may be affected; this causes the numerical behavior in real time to differ from the behavior in simulation. To see an illustration of this effect, try running the demo model `c166_multitasking`, described in the next section. To disallow sub-rate overruns and ensure that this effect does not occur, you should set **Maximum number of concurrent sub-rate overruns** to zero.

---

**Note** If the option "Maximum number of concurrent sub-rate overruns" is set to a value greater than zero, then the behavior of any Rate-Transition blocks may be affected. Specifically, if the model contains a Rate Transition block where the option "Ensure deterministic data transfer (maximum delay)" is selected, then this setting may not be honored.

---

# Multitasking Demo Model

The demo model `c166_multitasking` illustrates both execution profiling and the preemptive multitasking scheduler with configurable overrun handling.

The model is multirate, having tasks running at 1 ms, 4 ms, and 16 ms. It is configured to use the preemptive multitasking scheduler.

A special feature of this model is that each task is designed to perform an increasing number of calculations to increase the processor loading until that task reaches a target turnaround time. This behavior ensures that task overruns occur to demonstrate the behavior of the model in this situation.

Each block in the model, labeled `Load base rate`, `Load sub-rate 1`, `Load sub-rate 2` performs calculations, the result of which should always be 1 both in simulation and in real time. Any other result is a failure and should never occur.

The `Test Rate Interaction` blocks are designed to test whether data is transferred between tasks in a deterministic manner. In simulation, the output of each of these blocks is always zero, indicating that there is no drift between tasks running at different rates. When running in real time, under normal circumstances, the output is also zero; in this case the real-time behavior is deterministic and exactly matches the results in simulation. Even if task preemption and base-rate overruns occur, the output of these blocks will be zero so that the real-time behavior faithfully reproduces the results in simulation. The circumstance under which drift occurs is if sub-rate overruns occur during execution in real time; if this behavior is not desired, you should disallow sub-rate overruns by setting the maximum allowed number of sub-rate overruns to zero in the **C166 Options (1)** section in the Configuration Parameters dialog (see "Task Scheduler Overrun Options" on page 5-6).

You can double-click the block provided in the model to switch between profiling over serial or CAN connections.

## Running the Multitasking Demo

**1** Open the model by typing at the command line

c166_multitasking

If viewing in the Help browser, you can click the link to open the model. If you update the diagram you can see the sample-time colors.
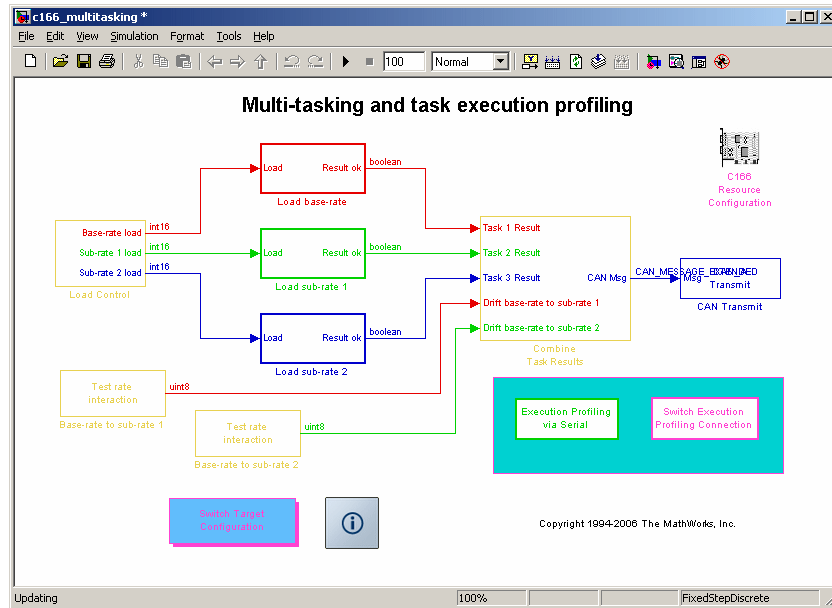


**2** Select **Simulation > Configuration Parameters**.

The Configuration Parameters dialog appears.

**3** Select **Link for TASKING** in the tree and change the **Build action** to Create, Build and Execute Application Project. Click **OK** to dismiss the dialog box.

**4** Make sure the target is connected to the host PC via serial or CAN cable. The default setting in this demo model is serial. You can double-click the Switch Execution Profiling Connection block to toggle between blocks for serial and CAN. See below for instructions if using CAN.

**5** To build and run the model, select the model window, and then press **Ctrl+B**.

Watch the messages in the command window as code is generated and loaded into the TASKING EDE, then the CrossView Pro Debugger starts, connects to the target, and downloads the code.

**6** In the CrossView window, click **Run** in the toolbar to start the application running on the target.

**7** At the command line, type

```
profile_c166 ('serial')
```

You will see messages in the command window as `profile_c166` runs.

When the data has been obtained the Help browser and a figure window appear, displaying the HTML report and the task execution profile.

**8** Scroll to view the HTML report on task timings and use the controls to zoom in on the MATLAB graphic to examine the details of the task overruns.

If using CAN, be sure to use CAN channel 0 (not 1) on the PC. You can double-click the Switch Execution Profiling Connection block in the model to switch to CAN, and follow the same instructions as for a serial connection, except step 7 when the application is running. At the command line, type

```
profile_c166 ('CAN')
```

You will see command line messages as the function tests the CAN channel, and requests and collects profiling data. When using CAN, it can be useful to run a monitor program such as `btest32` to verify that the model is running — for example you will see messages appearing on the CAN bus and you can see that you have connected the correct CAN channel.

## Interpreting the MATLAB Graphic

Dark shaded areas show the region where a task is executing. Light shaded areas show the region where a task is preempted by a higher priority task or ISR. Triangles indicate the beginning of a task. An example is shown following.
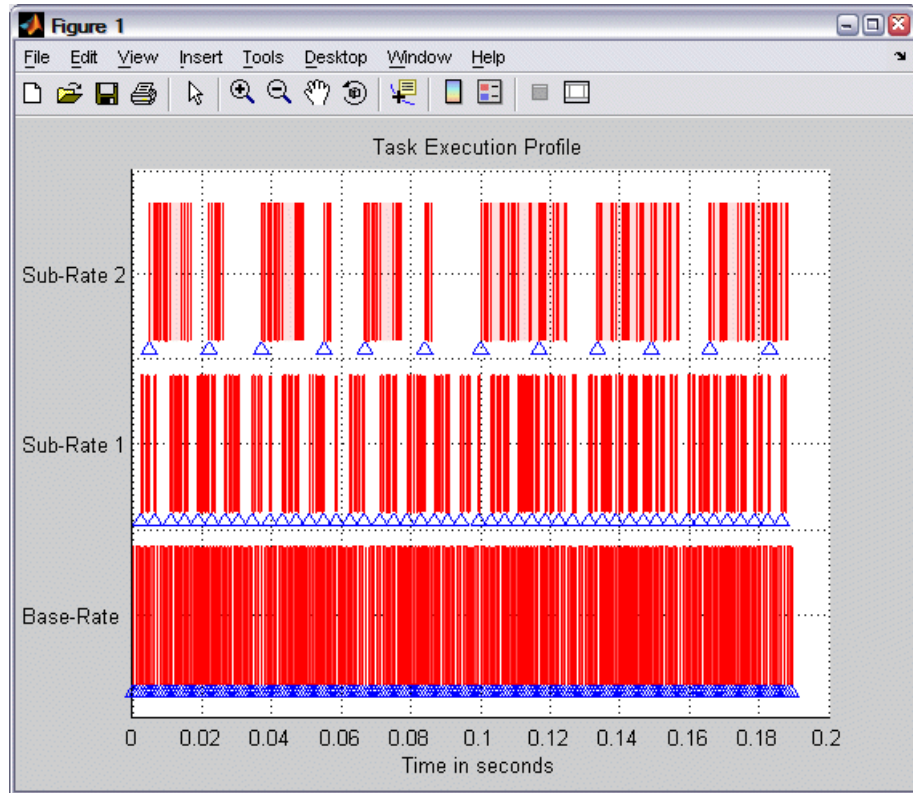
Zoom in to see the details of times that tasks are executing and being preempted, as shown in the following example.

.
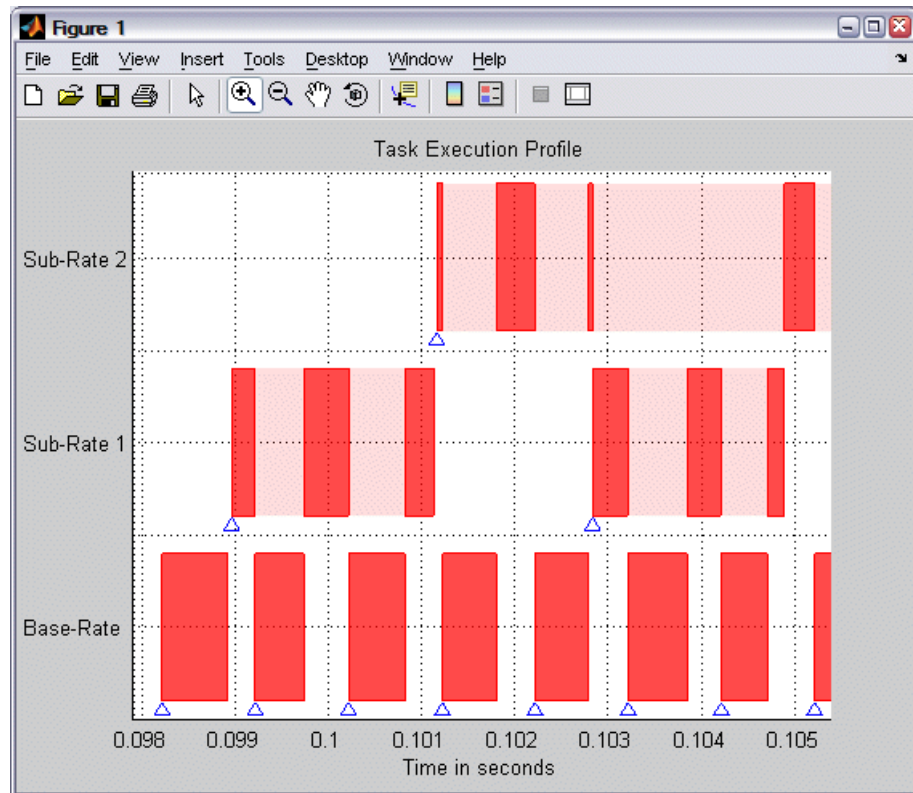
### The Generated HTML Report

See "Definitions" on page 5-4 for the terms task turnaround time, task execution time, and concurrent task overruns.

All times are in seconds. The timer resolution is 4e-007 seconds and the measurement range is 0.026214 seconds.

The report contains the following information:

• Worst-case task turnaround times

- Maximum task turnaround time for each task since model execution started. Note that the maximum task turnaround time that can be measured is limited by the timer measurement range.

- Maximum number of overruns for each task

  - Maximum number of concurrent task overruns since model execution started

- Analysis of recorded profiling data

  - Analysis of task turnaround times and task execution times based on recorded data over a period of 0.18139 second

**6**

# Blocks — By Category

# C166 Drivers

| | |
|---|---|
| Top-Level Blocks (p. 6-2) | Resource configuration for C166 microcontrollers |
| Asynchronous/Synchronous Serial Interface (p. 6-2) | Serial transmit and receive |
| CAN Interface (p. 6-3) | Controller Area Network (CAN) utilities |
| Execution Profiling (p. 6-3) | Configure execution profiling over CAN, TwinCAN, or serial connection |
| TwinCAN Interface (p. 6-3) | Controller Area Network (CAN) utilities for XC16x |
| Interrupts (p. 6-4) | Generate function-call triggers on interrupt |
| Utilities (p. 6-4) | Configure for predefined hardware configurations |
| Digital Input/Output (p. 6-4) | Configure digital input/output |

## Top-Level Blocks

| | |
|---|---|
| C166 Resource Configuration | Support device configuration for C166 microcontrollers |

## Asynchronous/Synchronous Serial Interface

| | |
|---|---|
| Serial Receive | Configure C166 microcontroller for serial receive |
| Serial Transmit | Configure C166 microcontroller for serial transmit |

## CAN Interface

| | |
|---|---|
| CAN Bus Status | Output Bus Off or Error Warning state of CAN module |
| CAN Calibration Protocol (C166) | Implement CAN Calibration Protocol (CCP) standard |
| CAN Receive | Receive CAN messages from CAN module on Infineon C166 microprocessor |
| CAN Reset | Reset CAN module |
| CAN Transmit | Transmit CAN messages via CAN module on Infineon C166 |

For information about CAN message blocks and CAN drivers, see the "CAN Blockset Reference".

## Execution Profiling

| | |
|---|---|
| C166 Execution Profiling via ASC0 | Provide serial interface to execution profiling engine |
| C166 Execution Profiling via CAN A | Provide CAN interface to execution profiling engine via CAN channel A |
| C166 Execution Profiling via TwinCAN A | Provide CAN interface to execution profiling engine via TwinCAN channel A for XC16x variants of Infineon C166 |

## TwinCAN Interface

| | |
|---|---|
| CAN Calibration Protocol (C166, TwinCAN) | Implement CAN Calibration Protocol (CCP) standard for XC16x variants of Infineon C166 |
| TwinCAN Bus Status | Output Bus Off or Error Warning state of a CAN node on XC16x variants of Infineon C166 |

| TwinCAN Receive | Receive CAN messages via TwinCAN module on XC16x variants of Infineon C166 |
| --- | --- |
| TwinCAN Reset | Reset CAN node on XC16x variants of Infineon C166 |
| TwinCAN Transmit | Transmit CAN messages from TwinCAN module on XC16x variants of Infineon C166 |

## Interrupts

| Fast External Interrupt | Generate asynchronous function-call trigger when interrupt occurs |
| --- | --- |

## Utilities

| Switch Target Configuration | Configure model and Target Preferences to one of a set of predefined hardware configurations |
| --- | --- |

## Digital Input/Output

| Digital In | Digital input driver that reads value of specified port or pin number |
| --- | --- |
| Digital Out | Digital output driver that sets logical state of specified pin |

# CAN Message Blocks and CAN Drivers

For information about CAN message blocks and CAN drivers, see the "CAN Blockset Reference".

# Blocks — Alphabetical List

# C166 Execution Profiling via ASC0

**Purpose**        Provide serial interface to execution profiling engine

**Library**        Embedded Target for Infineon C166 Microcontrollers/ C166 Driver
Library/ Execution Profiling
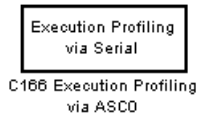
**Description**     The C166 Execution Profiling via ASC0 block provides a serial interface
to the execution profiling engine. On receipt of a start command
message, logging of execution profile data begins. On completion of a
logging run, the recorded data is automatically returned via the serial
interface (ASC0). See also the MATLAB command `profile_c166`.

```
Execution Profiling
    via Serial

C166 Execution Profiling
    via ASC0
```

`profile_c166('serial')` collects and displays execution profiling
data from a C166 target microcontroller that is running a suitably
configured application generated by Embedded Target for Infineon
C166 Microcontrollers. The `connection` may be set to `'serial'` in
order to collect data via a serial connection between the target and
the host computer.

The data collected is unpacked and then displayed in a summary HTML
report and as a MATLAB graphic.

See "The Profiling Command" on page 5-3 for instructions for setting
the bit rate automatically or manually.

To configure a model for use with execution profiling, you must perform
the following steps:

**1** Check the appropriate option in the **Target Specific Options** tab
of the Real-Time Workshop Options dialog.

**2** Make sure the model includes a C166 Execution Profiling block that
provides an interface between the target-side profiling engine, and
the host-side computer from which this command is run.

For more information, see Chapter 5, "Execution Profiling" which
includes instructions for the example demo `c166_multitasking`.

**Dialog Box**



**Block Parameters: C166 Execution Profiling via ASC0**

C166 Execution Profiling via Serial Port ASC0 (mask) (link)

Provides a serial interface to the execution profiling engine. On receipt of a start command to the serial port, logging of execution profile data is commenced. On completion of a logging run, the recorded data is automatically returned via serial.

No other serial blocks may be used in the model if an Execution Profiling via Serial block is present.

See also MATLAB command profile_c166.

Parameters

Sample time:

1

| OK | Cancel | Help | Apply |

**Sample time**

The sample time of the block. The faster the sample time of the block, the faster data will be uploaded at the end of the execution profiling run. You may want to run this block slower than the fastest rate in the system because the execution profiling itself imposes some loading on the processor. You can minimize this extra loading by not running it at the fastest rate.

# C166 Execution Profiling via CAN A

**Purpose**　　　Provide CAN interface to execution profiling engine via CAN channel A

**Library**　　　Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ Execution Profiling

**Description**

```
┌─────────────┐
│  Execution  │
│  Profiling  │
└─────────────┘
C166 Execution Profiling
      via CAN A
```

The C166 Execution Profiling via CAN A block provides a CAN interface to the execution profiling engine. On receipt of a start command message, logging of execution profile data begins. On completion of a logging run, the recorded data is automatically returned via CAN. You must specify the message identifiers for the start command and the returned data. These identifiers must be compatible with the values used by the host-side part of the execution profiling utility. See also the MATLAB command `profile_c166`.
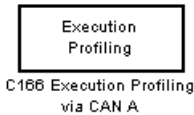
`profile_c166(connection)` collects and displays execution profiling data from a C166 target microcontroller that is running a suitably configured application generated by Embedded Target for Infineon C166 Microcontrollers. The `connection` may be set to `'CAN'` in order to collect data via a CAN connection between the target and the host computer. To use the CAN connection, you must have suitable CAN hardware installed on the host computer. This function tests for availability of CanCardX 1 or CanAc2Pci1 and defaults to a bit rate of 500K bits per second. If you need to use a different configuration, you should make a copy of this file (with a different name) and change the configuration data as required. The data collected is unpacked then displayed in a summary HTML report and as a MATLAB graphic.
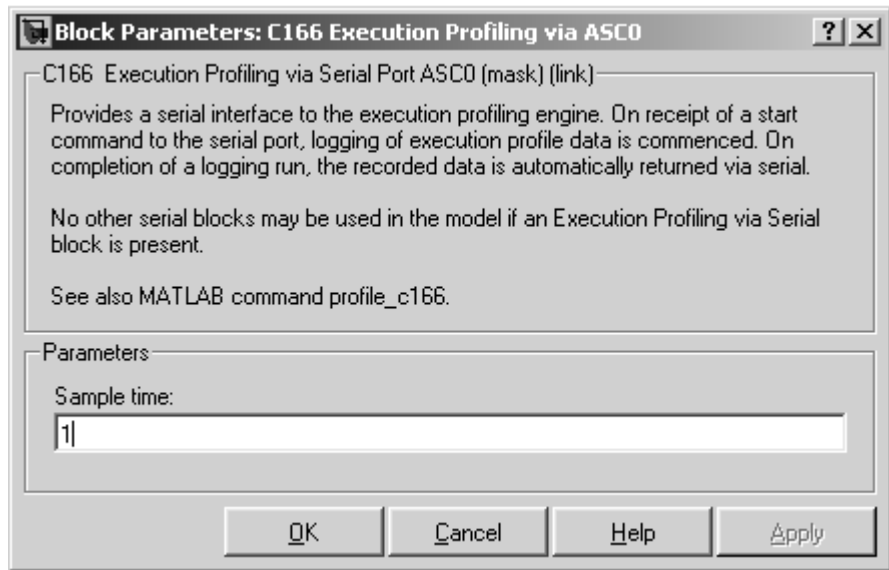
See "The Profiling Command" on page 5-3 for instructions for setting the bit rate automatically or manually.

To configure a model for use with execution profiling, you must perform the following steps:

**1** Check the appropriate option in the **Target Specific Options** tab of the Real-Time Workshop Options dialog.

**2** Make sure the model includes a C166 Execution Profiling block that provides an interface between the target-side profiling engine, and the host-side computer from which this command is run.

For more information, see Chapter 5, "Execution Profiling" which includes instructions for the example demo `c166_multitasking`.

**Dialog Box**

Block Parameters: C166 Execution Profiling via CAN A

C166 Execution Profiling via CAN Channel A (mask) (link)

Provides a CAN interface to the execution profiling engine. On receipt of a start command message, logging of execution profile data is commenced. On completion of a logging run, the recorded data is automatically returned via CAN.

You must specify the message identifiers for the start command and the returned data. These identifiers must be compatible with the values used by the host-side part of the execution profiling utility. See also MATLAB command profile_c166.

Parameters

Start command CAN message identifier:

hex2dec('1FFFFF00')

Returned data CAN message identifier:

hex2dec('1FFFFF01')

Sample time:

1

OK    Cancel    Help    Apply

**Start command CAN message identifier**
Set the identifier of the message to start logging execution profiling data. You should use the default unless you have modified `profile_c166`. This identifier must be compatible with

the values used by the host-side part of the execution profiling utility (`profile_c166`).

The utility `profile_c166` provides a mechanism for initiating an execution profiling run and for uploading the recorded data to the host machine. To perform this procedure using a CAN connection between host and target, `profile_c166` first sends a CAN message that is a command to start an execution profiling run. The CAN identifier for this message must be specified as the same value on the target as on the host. The host-side values are hard-coded in `profile_c166`. If you are using an unmodified version of the host-side utility, you should use the default value for this CAN message identifier. These are visible to help you avoid using the same identifier for other tasks.

**Returned data CAN message identifier**

Set the message identifier for the returned data. As with the message identifier for the start command, the value specified here must be the same as the hard-coded value in `profile_c166`.

**Sample time**

The sample time of the block. The faster the sample time of the block, the faster data will be uploaded at the end of the execution profiling run. You may want to run this block slower than the fastest rate in the system because the execution profiling itself imposes some loading on the processor. You can minimize this extra loading by not running it at the fastest rate.

**Purpose**      Provide CAN interface to execution profiling engine via TwinCAN channel A for XC16x variants of Infineon C166

**Library**      Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ Execution Profiling

**Description**  The C166 Execution Profiling via TwinCAN A block is for the TwinCAN interface and performs the same functions as the C166 Execution Profiling via CAN A block. For block parameter descriptions, see the C166 Execution Profiling via CAN A reference page.

Execution Profiling
via CAN
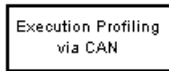
C166 Execution Profiling
via TwinCAN A

# C166 Resource Configuration

**Purpose**   Support device configuration for C166 microcontrollers

**Library**   Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library

**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 C166 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 7-10 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.

---

When the C166 Resource Configuration block is placed into a model, it modifies the `preloadfcn` callback of the model. If you wish to add a command to the `preloadfcn` callback of a model that already has an C166 Resource Configuration block, do not remove the commands that are already installed. Instead, copy the installed `preloadfcn` callback and append your commands. Then set the `preloadfcn` to the merged command. If you corrupt the `preloadfcn`, you can retrieve the command from any model that has a C166 Resource Configuration block, as the `preloadfcn` will be the same for all models. You can retrieve the `preloadfcn` with the following command: `plf = get_param(bdroot,'preloadfcn')`

### 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 Resource Configuration

- 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

- CAN Configuration Parameters: parameters for CAN interrupt levels

**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. Similarly, if you add CAN blocks to a model, a CAN configuration is created.

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 lost from the C166 Resource Configuration window when the model is saved and reopened. You can reactivate a configuration by simply adding an appropriate block into the model.

### 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`.
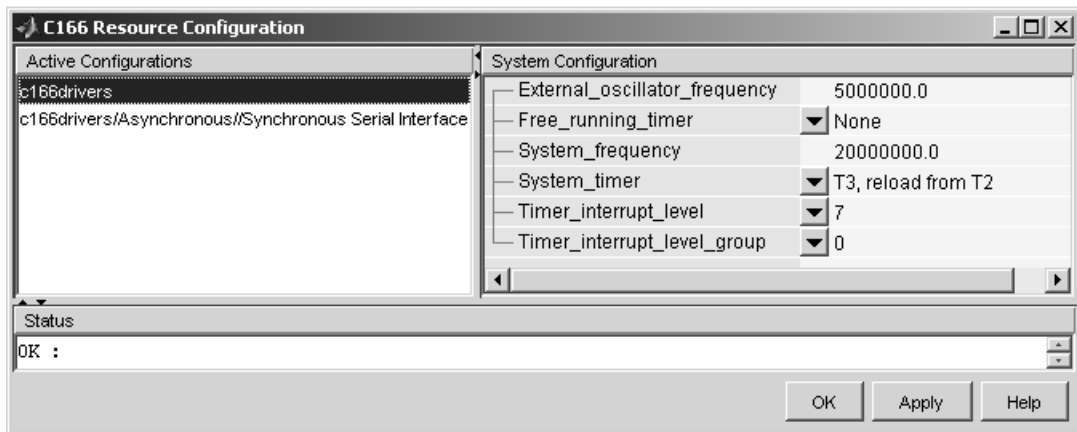
The C166 Resource Configuration window consists of the following elements:

- **Active Configurations** panel: This panel displays a list of currently active configurations. To edit a configuration, click 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 its entry in the list. From the menu that appears, select **Go to library**.

  To see documentation associated with an active configuration, right-click its entry in the list. From the 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 7-12.

> **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 following sections 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, read the relevant sections of the C166 User's Manual. You can find this document at the Infineon Web site at the following URL:

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

**C166 System Configuration Parameters**
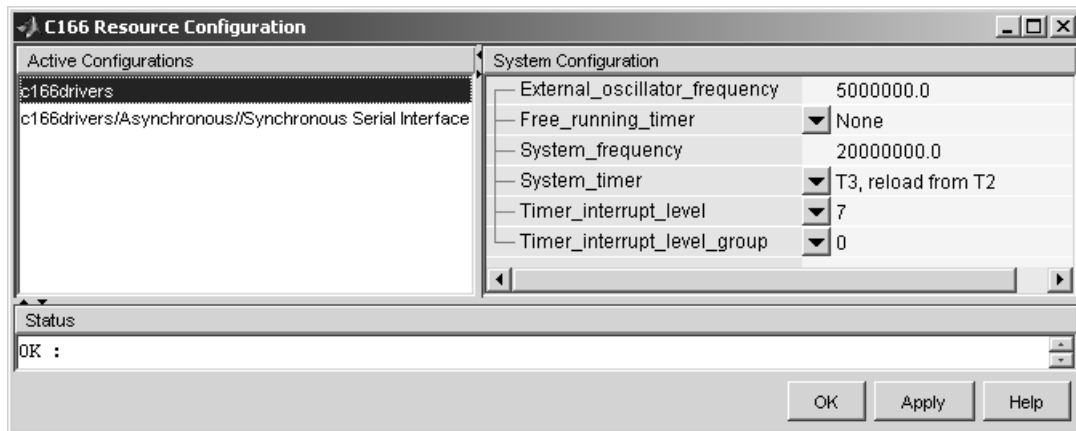


**External_oscillator_frequency**

Depending on your hardware variant, the Real-Time Clock (RTC) may be driven directly by the external oscillator input and it is,

therefore, important that the external oscillator frequency is set correctly. Otherwise, if the RTC is used to provide any timing services, the behavior will be incorrect. The default value for external oscillator frequency is 5 MHz. You should check your hardware manual to establish the correct value for your setup. Note you can choose the RTC as a System_timer, see below.

**Free_running_timer**

This parameter allows one of the on-chip timers to be configured for use with execution profiling. The selected timer is configured to run indefinitely at a known frequency and is used by the execution profiling engine to record the times at which tasks start or finish executing. See Chapter 5, "Execution Profiling" for more details.

**System_frequency**

You must set the system frequency of your C166 microcontroller hardware here. Note that the value depends on your hardware type and configuration. If you choose an incorrect value the model will 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 with no long term drift caused by higher priority interrupts. If you choose a single timer, e.g., T2 or RTC, 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**
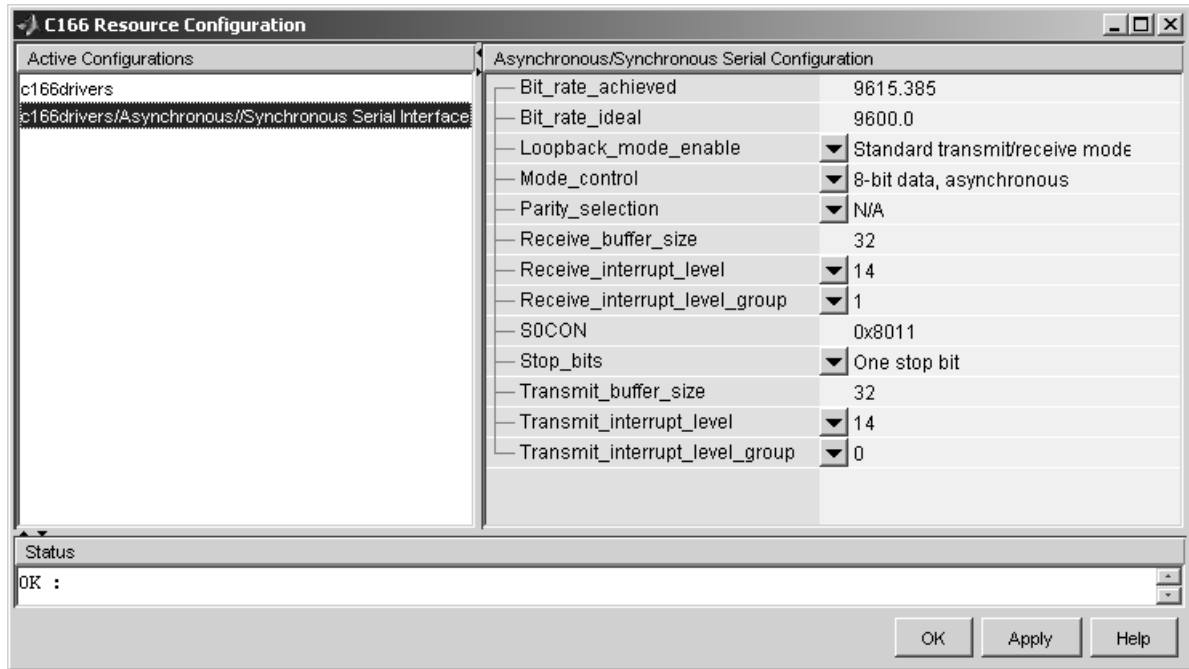


**Bit_rate_achieved**

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 are 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 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 allow only interrupt levels 14 and 15 to be used. The reason for this is that the drivers use the peripheral event controller (PEC), 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**
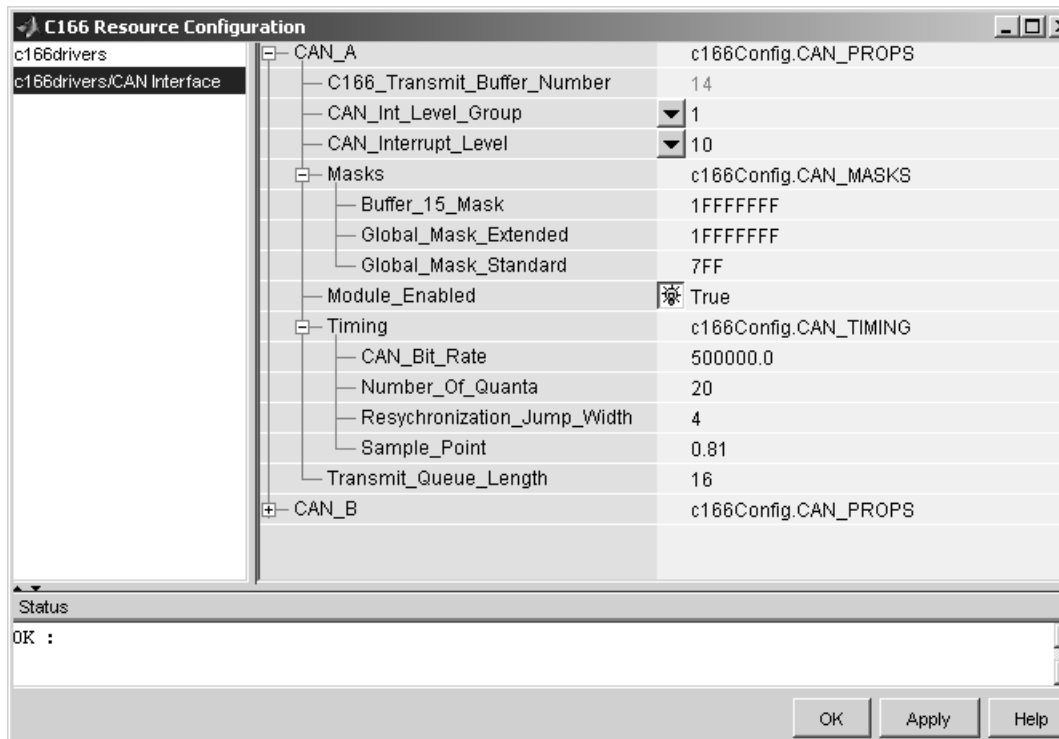> You must select either 1 or 2 stop bits.

**Transmit_buffer_size**
> See `Receive_buffer_size`.

**Transmit_interrupt_level** and **Transmit_interrupt_level_group**
> See `Receive` parameters above.

**CAN Configuration Parameters**



The parameters listed below are the same for CAN modules A and B.

**C166_Transmit_Buffer_Number**
    This parameter is read only; all transmitted messages are sent
    from buffer 14.

**CAN_Int_Level_Group** and **CAN_Interrupt_Level**
    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. Note that CAN module interrupts must be set to a higher priority than timer interrupts. Use the **Validate Configuration** button to make sure you do not select an interrupt level that is already in use.

## Masks

You can use these mask configuration parameters to choose to ignore certain bits. In general, a CAN message is received only if its identifier is an exact match with the identifier specified in one of the receive buffers. You can use mask parameters to indicate that some of the bits in the received message identifier are "don't care."

### Buffer_15_Mask

This mask applies to buffer 15 only. Each bit in the mask that is set to zero causes the corresponding bit in the received message identifier to be ignored when comparing it to the message identifier that buffer 15 is configured to receive.

### Global_Mask_Extended

This mask applies to any of buffers 1 to 14 that are configured to receive messages with an extended identifier. Each bit in the mask that is set to zero causes the corresponding bit in the received message identifier to be ignored when comparing it to the message identifier that this buffer is configured to receive.

### Global_Mask_Standard

This mask applies to any of buffers 1 to 14 that are configured to receive messages with an standard identifier. Each bit in the mask that is set to zero causes the corresponding bit in the received message identifier to be ignored when comparing it to the message identifier that this buffer is configured to receive.

### Module_Enabled

If the module is enabled, then initialization code for that CAN module is generated. Use this setting to prevent generation of driver code for a CAN module that is not required, or not available on your hardware variant.

# C166 Resource Configuration

**Timing**

**CAN_Bit_Rate**
> Enter the desired bit rate. The default bit rate is 500000.

**Number_Of_Quanta**
> The number of CAN module clock ticks per message bit.

**Resynchronization_Jump_Width**
> The maximum number of clock ticks that the CAN device can resynchronize over when it detects that it is losing message synchronization.

**Sample_Point**
> The point in the message where the CAN module samples the value of the message bit.

**Transmit_Queue_Length**
> Length (number of messages) of the transmit queue. The transmit queue holds messages that are waiting to be transmitted. An increase in performance can be achieved by reducing the queue length. However, if the queue's length is too small, it may become full, causing messages to be lost.
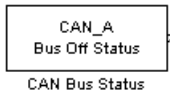
**Purpose**  Output Bus Off or Error Warning state of CAN module

**Library**  Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ CAN Interface

**Description**  The CAN Bus Status block provides an indicator of the state of the selected CAN module. The block has a single output that may be set to indicate either the Bus Off or Error Warning state of the module.

```
CAN_A
Bus Off Status

CAN Bus Status
```

**Dialog Box**
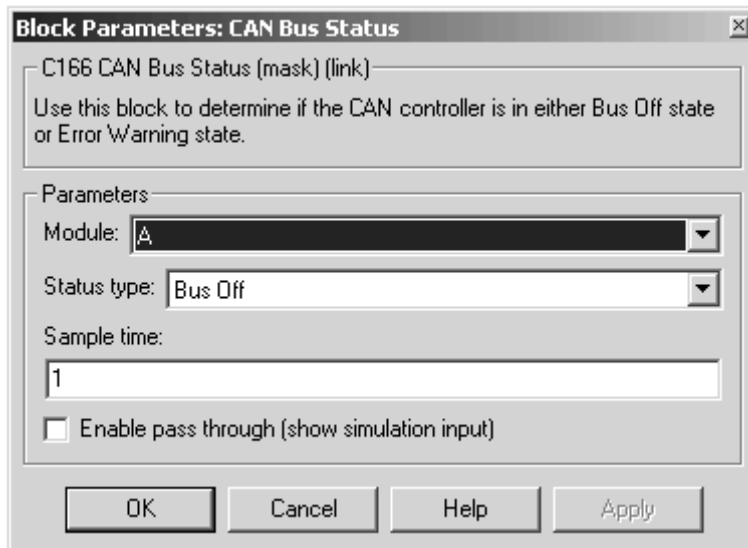


**Module**
Select CAN module A or B.

**Status type**
Choose Bus Off or Error Warning.

# CAN Bus Status

**Sample time**
  The sample time of this block.

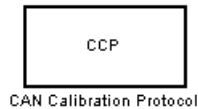**Purpose**      Implement CAN Calibration Protocol (CCP) standard

**Library**      Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ CAN Interface

**Description**

```
    CCP

CAN Calibration Protocol
```

The CAN Calibration Protocol (C166) block provides an implementation of a subset of the CAN Calibration Protocol (CCP) Version 2.1. CCP is a protocol for communicating between the target processor and the host machine over CAN. In particular, a calibration tool (see "Compatibility with Calibration Packages" on page 7-26) running on the host can communicate with the target, allowing remote signal monitoring and parameter tuning.

This block processes a Command Receive Object (CRO) and outputs the resulting Data Transmission Object (DTO) and Data Acquisition (DAQ) messages.

For more information on CCP, refer to *ASAM Standards: ASAM MCD: MCD 1a* on the Association for Standardization of Automation and Measuring Systems (ASAM) Web site at `http://www.asam.de`.

### Using the DAQ Output

The DAQ output is the output for any CCP Data Acquisition (DAQ) lists that have been set up. You can use the ASAP2 file generation feature of the Real-Time (RT) target to

- Set up signals to be transmitted using CCP DAQ lists.

- Assign signals in your model to a CCP event channel automatically (see "Generating ASAP2 Files" on page 2-17).

Once these signals are set up, event channels then periodically fire events that trigger the transmission of DAQ data to the host. When this occurs, CAN messages with the appropriate CCP/DAQ data appear on the DAQ output, along with an associated function call trigger.

The calibration tool (see "Compatibility with Calibration Packages" on page 7-26) must use CCP commands to assign an event channel and data to the available DAQ lists, and interpret the synchronous response.
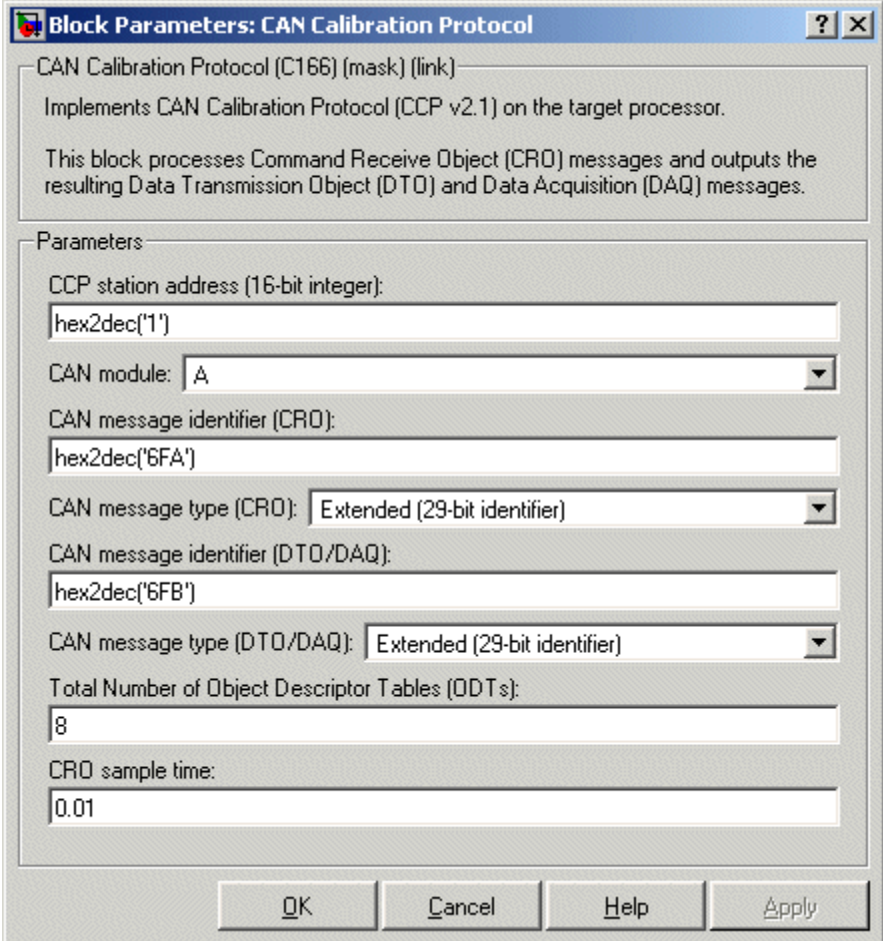
Using DAQ lists for signal monitoring has the following advantages over the polling method:

- There is no need for the host to poll for the data. Network traffic is halved.

- The data is transmitted at the correct update rate for the signal. Therefore, there is no unnecessary network traffic generated.

- Data is guaranteed to be consistent. The transmission takes place after the signals have been updated, so there is no risk of interruptions while sampling the signal.

**Note** The Embedded Target for Infineon C166 Microcontrollers does not currently support event channel prescalers.

**Dialog Box**

Block Parameters: CAN Calibration Protocol    ? ✕

CAN Calibration Protocol (C166) (mask) (link)

Implements CAN Calibration Protocol (CCP v2.1) on the target processor.

This block processes Command Receive Object (CRO) messages and outputs the resulting Data Transmission Object (DTO) and Data Acquisition (DAQ) messages.

Parameters

CCP station address (16-bit integer):

hex2dec('1')

CAN module: A

CAN message identifier (CRO):

hex2dec('6FA')

CAN message type (CRO): Extended (29-bit identifier)

CAN message identifier (DTO/DAQ):

hex2dec('6FB')

CAN message type (DTO/DAQ): Extended (29-bit identifier)

Total Number of Object Descriptor Tables (ODTs):

8

CRO sample time:

0.01

OK    Cancel    Help    Apply

**CAN station address (16 bit integer)**
> The station address of the target. The station address is interpreted as a uint16. It is used to distinguish between different targets. By assigning unique station addresses to targets sharing the same CAN bus, it is possible for a single host to communicate with multiple targets.

# CAN Calibration Protocol (C166)

**CAN module**
Choose CAN module A or B.

**CAN message identifier (CRO)**
Specify the CAN message identifier for the Command Receive Object (CRO) message you want to process.

**CAN message type (CRO)**
The incoming message type. Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

**CAN message identifier (DTO/DAQ)**
The message identifier is the CAN message ID used for Data Transmission Object (DTO) and Data Acquisition (DAQ) message outputs.

**CAN message type (DTO/DAQ)**
The message type to be transmitted by the DTO and DAQ outputs. Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

**Total number of Object Descriptor Tables (ODTs)**
The default number of Object Descriptor Tables (ODTs) is 8. These ODTs are shared equally between all available DAQ lists. You can choose a value between 0 and 254, depending on how many signals you wish to log simultaneously. You must make sure you allocate at least 1 ODT per DAQ list, or your build will fail. The calibration tool will give an error message if there are too few ODTs for the number of signals you specify for monitoring. Be aware that too many ODTs can make the sample time overrun. If you choose more than the maximum number of ODTs (254), the build will fail.

A single ODT uses 56 bytes of memory. Using all 254 ODTs would require over 14 KB of memory, a large proportion of the available memory on the target. To conserve memory on the target, the default number is low, allowing DAQ list signal monitoring with reduced memory overhead and processing power.

As an example, if you have five different rates in a model, and you are using three rates for DAQ, then this will create three DAQ lists and you must make sure you have at least three ODTs. ODTs are shared equally among DAQ lists and, therefore, you will end up with one ODT per DAQ list. With less than three ODTs, you get zero ODTs per DAQ list and the behavior is undefined.

Taking this example further, say you have three DAQ lists with one ODT each, and start trying to monitor signals in a calibration tool. If you try to assign too many signals to a particular DAQ list (that is, signals requiring more space than seven bytes (one ODT) in this case), then the calibration tool will report this as an error.

**CRO sample time**
The sample time for CRO messages.

## Supported CCP Commands

The following CCP commands are supported by the CAN Calibration Protocol (C166) block:

- `CONNECT`
- `DISCONNECT`
- `DNLOAD`
- `DNLOAD_6`
- `EXCHANGE_ID`
- `GET_CCP_VERSION`
- `GET_DAQ_SIZE`
- `GET_S_STATUS`
- `SET_DAQ_PTR`
- `SET_MTA`
- `SET_S_STATUS`

- SHORT_UP

- START_STOP

- START_STOP_ALL

- TEST

- UPLOAD

- WRITE_DAQ

### Compatibility with Calibration Packages

The above commands support

- Synchronous signal monitoring via calibration packages that use DAQ lists

- Asynchronous signal monitoring via calibration packages that poll the target

- Asynchronous parameter tuning via CCP memory programming

This CCP implementation has been tested successfully with the Vector-Informatik CANape calibration package running in both DAQ list and polling mode, and with the Accurate Technologies, Inc., Vision, calibration package running in DAQ list mode. (Note that Accurate Technologies, Inc., Vision does not support the polling mechanism for signal monitoring).
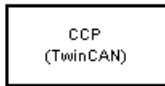
**Purpose**     Implement CAN Calibration Protocol (CCP) standard for XC16x variants of Infineon C166

**Library**     Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ CAN Interface

**Description**     The CAN Calibration Protocol (C166, TwinCAN) block is for the TwinCAN interface and performs the same functions as the CAN Calibration Protocol (C166) block. For block parameter descriptions, see the CAN Calibration Protocol (C166) reference page.

```
    CCP
 (TwinCAN)
```
CAN Calibration Protocol

# CAN Receive

**Purpose**     Receive CAN messages from CAN module on Infineon C166
                microprocessor

**Library**     Embedded Target for Infineon C166 Microcontrollers/ C166 Driver
                Library/ CAN Interface

**Description** The CAN Receive block receives CAN messages from a CAN module.
                The CAN Receive block can reserve one of the buffers on the CAN
                module. Alternatively, you can instruct the CAN Receive block to
                select a hardware buffer automatically from the available buffers. The
                CAN Receive block has two outputs: a data output and a function-call
                trigger output. The CAN Receive block polls its message buffer at a rate
                determined by the block's sample time. When the CAN Receive block
                detects that a message has arrived, the function-call trigger is activated.
                You should use a function-call subsystem, activated by the trigger, to
                decode the message available at the CAN Receive block data output.

**Dialog
Box**
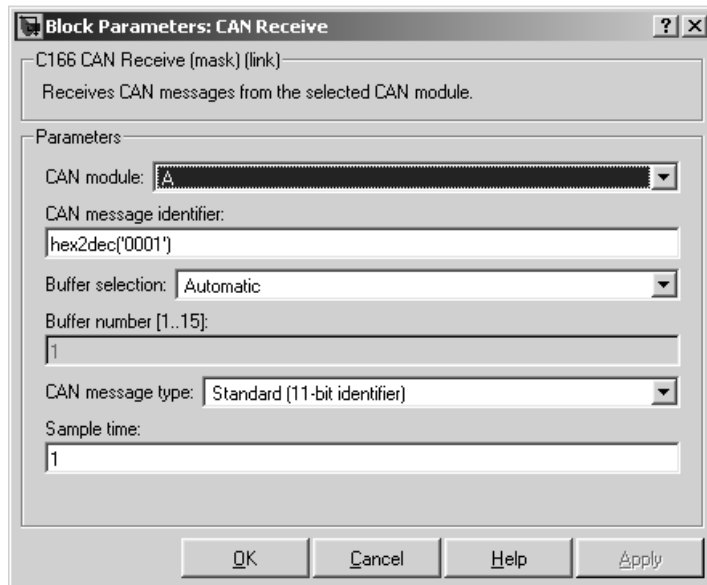
**CAN module**

Select CAN module `A` or `B`. The CAN modules can receive messages independently.

**CAN message identifier**

The identifier of the message you want to receive. Note that if you have set the CAN configuration parameters in your model to mask out certain bits (e.g., the message identifier field), you may receive messages with identifiers other than the identifier specified here. See "CAN Configuration Parameters" on page 7-16.

**Buffer selection**

Choose `Automatic` or `Manual`. When the automatic option is selected, the CAN Receive block automatically selects a receive buffer from the available buffers. Use this automatic buffer selection, unless you want to use buffer 15 with its individually programmable mask.

**Buffer number [1..15]**

This field is enabled if the **Buffer selection** is `Manual`. The buffer number specifies the identifier of the receive buffer for this block. Select `Automatic` buffer selection instead of manually specifying the buffer, unless you want to use buffer 15 with its individually programmable mask.

**CAN message type**

The type of message you want to receive. Select either `Standard(11-bit identifier)` or `Extended(29-bit identifier)`.

**Sample time**

Determines the rate at which to sample the buffer to see if a new message has arrived.

# CAN Receive

**Note** The CAN Receive block sample time must be set to a value that is smaller than the minimum time between CAN messages that will be received into the corresponding buffer. If more than one message is received into a buffer during a single sample interval, the older message will be overwritten.
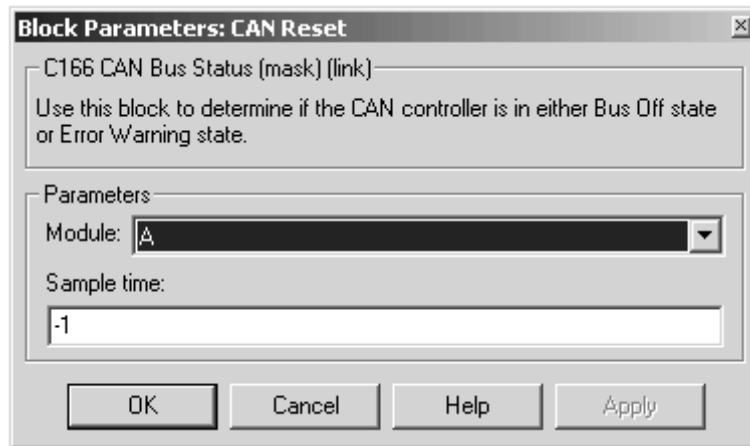
**Purpose**      Reset CAN module

**Library**      Embedded Target for Infineon C166 Microcontrollers/ C166 Driver
Library/CAN Interface

**Description**  The CAN Reset block reinitializes the CAN module. We recommend
that you place this block in a triggered subsystem, with a sample time
of -1 (inherited).

```
CAN_A
Reset

CAN Reset
```
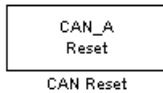
**Dialog Box**



**Module**
   Select CAN module A or B.

**Sample time**
   The sample time of this block.

# CAN Transmit

**Purpose**        Transmit CAN messages via CAN module on Infineon C166

**Library**        Embedded Target for Infineon C166 Microcontrollers/C166 Driver Library/CAN Interface

**Description**

> CAN_A
> Msg  Transmit
>
> CAN Transmit

The CAN Transmit block transmits a CAN message onto the CAN bus. Three modes of transmission are available with the CAN Transmit block.

The default mode is to use a priority-based message queue shared by all transmit blocks operating in this mode; the priority-based message queue operates with CAN buffer 14; when a message is successfully transmitted from this buffer, an interrupt is generated and the highest priority message from the queue is loaded into the hardware buffer ready to be transmitted. This mode has the advantage of allowing several messages with different identifiers to be transmitted without each message requiring a dedicated hardware buffer. Note that although messages are taken from the queue in order of priority, it is possible for a low priority message to be present in the hardware buffer and higher priority messages cannot then be transferred from the queue until transmission of the low priority message is complete.

The second transmit mode is to use a dedicated CAN buffer; in this case, messages to be transmitted are loaded directly into a CAN buffer that is used exclusively by the block. No queue is used, which means that in case the previous message has not been transmitted, it will be overwritten by the new one. This transmit mode does not use interrupts. An advantage of using the dedicated buffer mode is that there is reduced delay in transmitting high-priority messages, and reduced processor overhead that is otherwise required for queue management and servicing interrupts.

The third transmit mode is to use a First In First Out (FIFO) queue with dedicated buffer. In this mode, messages are placed in a queue and then transmitted on a first in, first out basis. This mode is useful if several messages, possibly with the same CAN identifier, must be transmitted in sequence; this may be a requirement if CAN is being used for data acquisition.

The CAN Transmit block should be connected to CAN Message Packing/Unpacking blocks. Do not ground the block or leave it unconnected.

**Dialog Box**
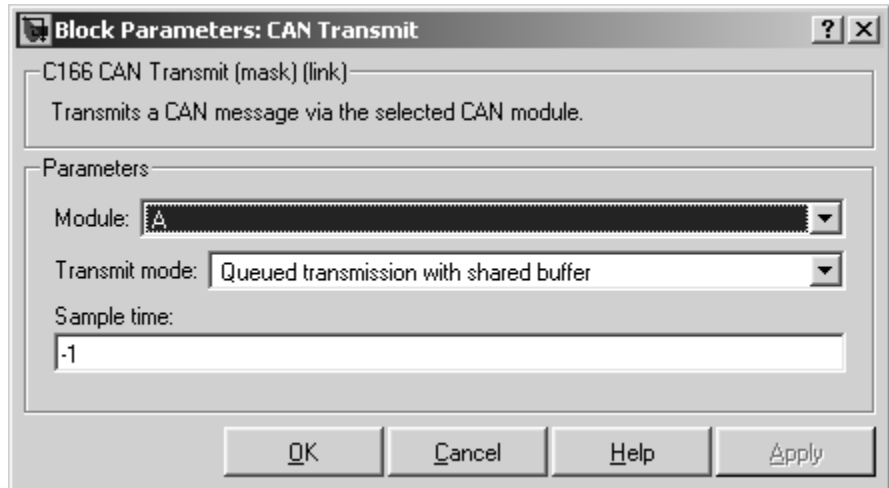


**Module**

Select CAN module A or B.. The CAN modules can receive messages independently.

**Transmit mode**

Select one of the three modes described above: queued transmission with shared buffer, direct transmission with dedicated buffer, or FIFO queue with dedicated buffer.

**Buffer selection**

Only for selecting dedicated buffers — available only if you select direct transmission or FIFO queue transmit modes. Choose either automatic or manual selection of the hardware buffer number.

**Buffer number**

This option is available only if the buffer selection is available and set to manual. You must select a buffer number between 1 and 14. Note if more than one message is ready to be transmitted,

then the one in the lower buffer number will be sent first. Select buffer numbers such that the higher the message priority, the lower the buffer number.

**Sample time**

Choose -1 to inherit the sample time from the driving blocks. The CAN Transmit block does not inherit constant sample times and runs at the base rate of the model if driven by invariant signals.

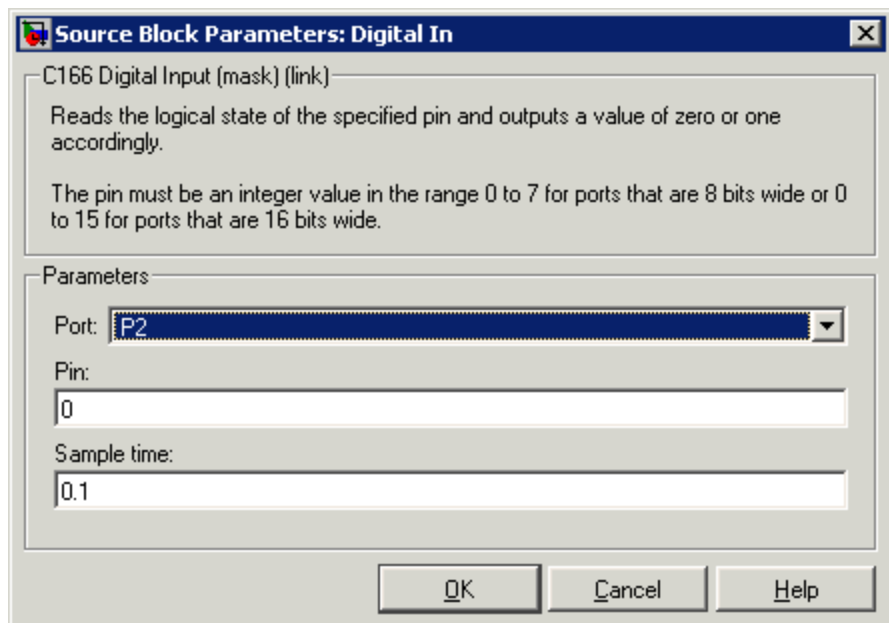**Purpose**        Digital input driver that reads value of specified port or pin number

**Library**        Embedded Target for Infineon C166 Microcontrollers/C166 Driver
                   Library/Digital Input/Output

**Description**

```
Digital In
on P2.0
──────────
Digital In
```
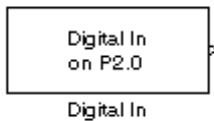
The Digital In block reads the logical state of the specified pin and
outputs a value of zero or one accordingly.

**Dialog
Box**

Source Block Parameters: Digital In                                    ✕

─C166 Digital Input (mask) (link)─────────────────────────────────────

Reads the logical state of the specified pin and outputs a value of zero or one
accordingly.

The pin must be an integer value in the range 0 to 7 for ports that are 8 bits wide or 0
to 15 for ports that are 16 bits wide.

─Parameters─────────────────────────────────────────────────────────

Port: P2                                                          ▼

Pin:

0

Sample time:

0.1

                              OK          Cancel          Help

# Digital In

**Port**

Select a port. Options are P0L, P0H, P1L, P1H, P2–P8.

**Pin**

The pin must be an integer value in the range 0 to 7 for ports that are 8 bits wide, or 0 to 15 for ports that are 16 bits wide.

**Sample time**

The time interval between samples. The default is 0.1. See "Specifying Sample Time" in the Simulink documentation for more information.

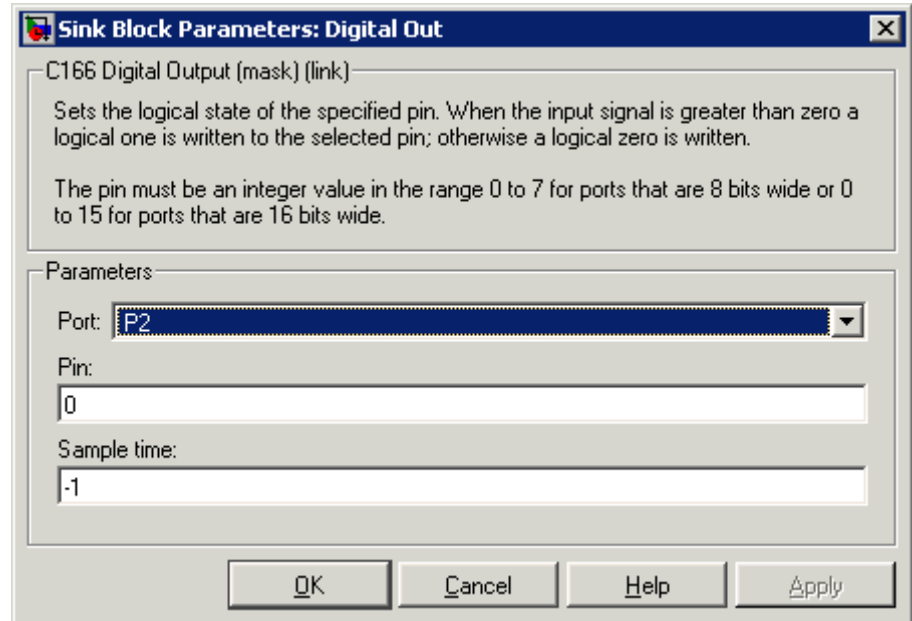**Purpose**     Digital output driver that sets logical state of specified pin

**Library**     Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ Digital Input/Output

**Description**     The Digital Out block sets the logical state of the specified pin according to the input signal. When the input signal is greater than zero, a logical one is written to the selected pin; otherwise a logical zero is written.

```
Digital Out
  on P2.0

Digital Out
```

**Dialog Box**

Sink Block Parameters: Digital Out ⊠

C166 Digital Output (mask) (link)

Sets the logical state of the specified pin. When the input signal is greater than zero a logical one is written to the selected pin; otherwise a logical zero is written.

The pin must be an integer value in the range 0 to 7 for ports that are 8 bits wide or 0 to 15 for ports that are 16 bits wide.

Parameters

Port: P2 ▼

Pin:
0

Sample time:
-1

| OK | Cancel | Help | Apply |
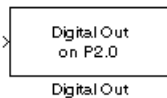
**Port**
Select a port. Options are P0L, P0H, P1L, P1H, P2–P8 (not P5).

# Digital Out

**Pin**

>The pin must be an integer value in the range 0 to 7 for ports that are 8 bits wide, or 0 to 15 for ports that are 16 bits wide.

**Sample time**

>The time interval between samples. The default is -1, inherited. See "Specifying Sample Time" in the Simulink documentation for more information.

**Purpose**     Generate asynchronous function-call trigger when interrupt occurs

**Library**     Embedded Target for Infineon C166 Microcontrollers/ C166 Driver
Library/ Interrupts

## Description

```
┌─────────────────┐
│  Fast External  │
│ Interrupt on P2.8│ ▷
└─────────────────┘
Fast External Interrupt
```
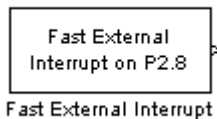
The Fast External Interrupt block executes a function-call triggered
subsystem in the context of the service routine for a fast external
interrupt. To generate the interrupt, you must select one of the upper
eight pins of Port 2 (P2.8 to P2.15)

The function-call subsystem will be executed as an asynchronous
task. Use this block to assign the task a Simulink priority and a CPU
interrupt level. The settings that you assign must be consistent with
priorities and interrupt levels of other tasks defined in the model.

**Limitations on XC16x Hardware**

On XC16x hardware, this block is unable to generate code to enable
fast external interrupts. Fast external interrupts must be enabled by
setting bits in the register EXICON. On XC16x devices this register is
write protected after execution of the special EINIT instruction by the
processor's register security mechanism. It is not possible for the driver
block to generate code that is executed before the EINIT instruction.

If you want to use this block on XC16x hardware, you must set the
required bits in register EXICON in the Project Options within the
Tasking EDE. For example, to enable fast external interrupts on rising
or falling edges for both of pins P2.8 and P2.9 (as required by the demo
model c166_async), follow these steps:

**1** Build the model c166_async.

# Fast External Interrupt

**2** Open the project `c166_async_c166` within the Tasking EDE.

**3** Select **Project > Project Options**.

**4** In the Project Options dialog, select in the tree **Application > Startup > EXICON**.

    **a** Set the value to `0x000F`.

    **b** Select the check box to **Include in startup code**.

**5** From within the Tasking EDE, re-build the project `c166_async_c166`.

**6** Download to the XC16x by launching Crossview from the CrossView button in the Tasking EDE.

Alternatively, you can create a new template project with the required setting for EXICON. You can easily create a new template project from the MATLAB **Start** menu by selecting **Start > Simulink > Link for TASKING > Create New Template Projects.**

**Dialog Box**

**Port 2 pin number**
    Select a port. Options are 8 to 15.

**Trigger mode**
    Select from `Rising or falling edge` (the default), `Rising edge`, `Falling edge`, or `Disabled`.

**Priority**
    Set a Simulink priority. The default is 30.

**Interrupt level**
    Select an interrupt level from 1 to 15. The default is 5.

**Interrupt level group**
    Select an interrupt level group from 0 to 3. The default is 1.

**Show simulation input**
    Select this check box (and click **Apply**) to get an input port for simulation.

**Purpose**        Configure C166 microcontroller for serial receive

**Library**        Embedded Target for Infineon C166 Microcontrollers/ C166 Driver 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 retrieves only those bytes that have already been received and placed in the internal buffer; it never waits 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 "Asynchronous/Synchronous Serial Interface Configuration Parameters" on page 7-14.

# Serial Receive

> **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 causes 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 2-12.

### 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 the value of input signal if supplied, or the width of output signal otherwise.

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

**Dialog Box**
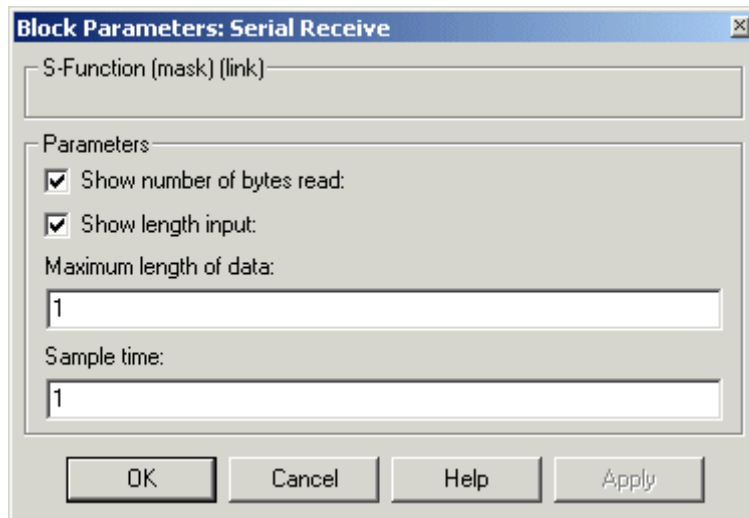


**Show number of bytes read**

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 7-14.

**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.

# Serial Transmit

**Purpose**

Configure C166 microcontroller for serial transmit

**Library**

Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ Asynchronous/Synchronous Serial Interface

**Description**
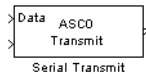


The Serial Transmit block transmits bytes over the C166 microcontroller Synchronous/Asynchronous Serial Interface ASC0. You can use it either to transmit a fixed number of bytes, or by enabling the second input, transmit a variable number of bytes each time this block is called.

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 "Asynchronous/Synchronous Serial Interface Configuration Parameters" on page 7-14.

**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 causes 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 2-12.

**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.

# Serial Transmit

**Block Parameters: Serial Transmit1** ✕

S-Function (mask) (link)

Parameters

Sample time:

-1

☑ Show length input:

☑ Show number of bytes sent

| OK | Cancel | Help | Apply |
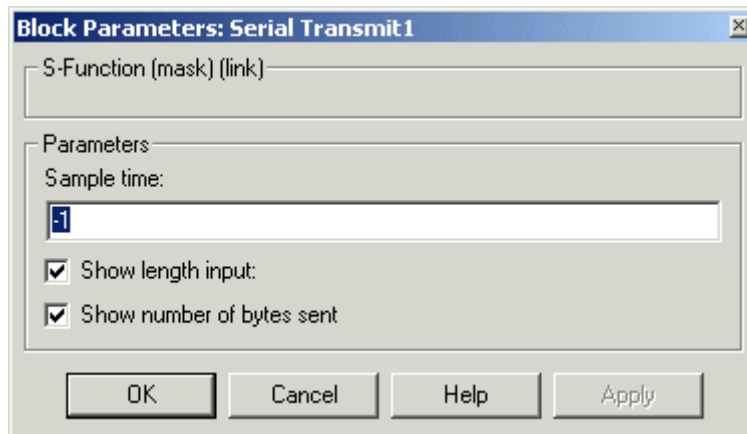
**Sample time**
> The time interval between samples. To inherit the sample time, leave this parameter at the default -1. See "Specifying Sample Time" in the Simulink documentation for more information.

**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.

**Purpose**     Configure model and Target Preferences to one of a set of predefined hardware configurations

**Library**     Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ Utilities

## Description



Switch Target
Configuration
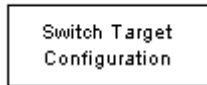
Switch Target
Hardware Configuration

Place the Switch Target Configuration block in your model and double-click it to run a convenience function that configures your model and Target Preferences to one of a set of predefined configurations. The Link for TASKING Option Set Selection dialog box appears, and you must choose a configuration for your processor type from the list. The suffixes '_hw' and '_sim' mean hardware or instruction set simulator.

# Switch Target Configuration
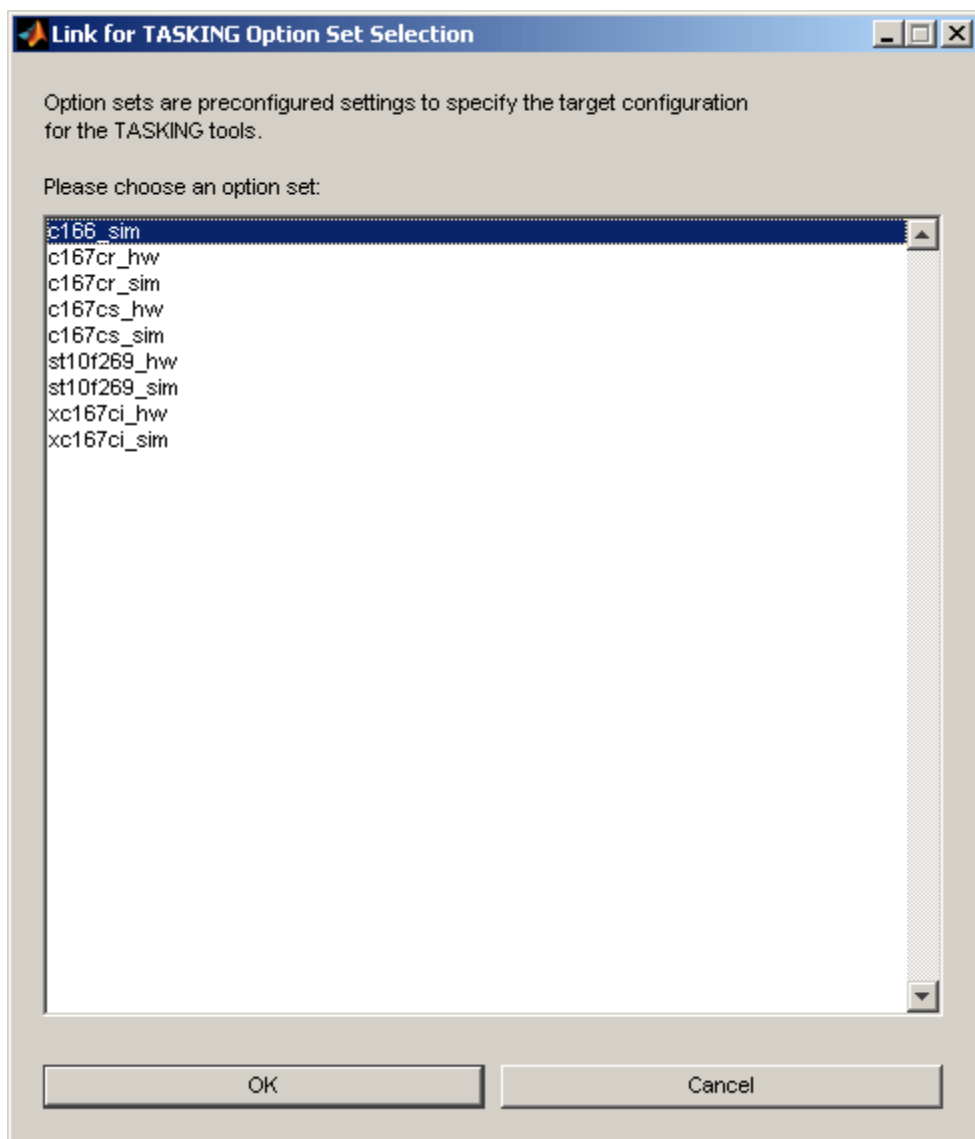


The predefined configurations include settings for

- Phytec phyCORE-167 ST10F269

- Phytec phyCORE-167 C167CS

- Phytec kitCON-167 C167CR

- Infineon XC167CI Starter Kit

# TwinCAN Bus Status

**Purpose**      Output Bus Off or Error Warning state of a CAN node on XC16x
variants of Infineon C166

**Library**      Embedded Target for Infineon C166 Microcontrollers/ C166 Driver
Library/ TwinCAN Interface

**Description**  The TwinCAN Bus Status block is for the TwinCAN interface and
performs the same functions as the CAN Bus Status block. For block
parameter descriptions, see the CAN Bus Status reference page.



TwinCAN_A
Bus Off Status

TwinCAN Bus Status

**Purpose**      Receive CAN messages via TwinCAN module on XC16x variants of Infineon C166

**Library**      Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ TwinCAN Interface

**Description**   The TwinCAN Receive block receives CAN messages from a TwinCAN module. The TwinCAN Receive automatically reserves one of the buffers on the TwinCAN module. The TwinCAN Receive block has two outputs: a data output and a function call trigger output. The TwinCAN Receive block polls its message buffer at a rate determined by the block's sample time. When the TwinCAN Receive block detects that a message has arrived, the function call trigger is activated. You should use a function call subsystem, activated by the trigger, to decode the message available at the TwinCAN Receive block data output.

This block has the same parameters as the CAN Receive block, except there is no option to `Automatically select buffer` or `Buffer number`. For block parameter descriptions, see the CAN Receive reference page.

# TwinCAN Reset

**Purpose**        Reset CAN node on XC16x variants of Infineon C166

**Library**        Embedded Target for Infineon C166 Microcontrollers/C166 Driver Library/TwinCAN Interface

**Description**    The TwinCAN Reset block is for the TwinCAN interface and performs the same functions as the CAN Reset block. For block parameter descriptions, see the CAN Reset reference page.

**Purpose**      Transmit CAN messages from TwinCAN module on XC16x variants of Infineon C166

**Library**      Embedded Target for Infineon C166® Microcontrollers/ C166 Driver Library/ TwinCAN Interface

**Description**  The TwinCAN Transmit block transmits a CAN message onto the CAN bus. Two modes of transmission are available with the CAN Transmit block, as described below.

The first transmit mode is to use a dedicated CAN buffer; in this case, messages to be transmitted are loaded directly into a CAN buffer that is used exclusively by the block. No queue is used, which means that in case the previous message has not been transmitted, it will be overwritten by the new one. This transmit mode does not use interrupts. An advantage of using the dedicated buffer mode is that there is minimal delay in transmitting high-priority messages.

The second transmit mode is to use a First In First Out (FIFO) queue with dedicated buffer. In this mode, messages are placed in a queue and then transmitted on a first in, first out basis. This mode is useful if several messages, possibly with the same CAN identifier, must be transmitted in sequence; this may be a requirement if CAN is being used for data acquisition.

The TwinCAN Transmit block should be connected to CAN Message Packing/Unpacking blocks. Do not ground the block or leave it unconnected.

# TwinCAN Transmit

**Dialog Box**

Block Parameters: TwinCAN Transmit

C166 TwinCAN Transmit (mask) (link)

Transmits a CAN message via the selected TwinCAN module.

Parameters

TwinCAN Node: A

Transmit mode: Direct transmission with dedicated buffer

Buffer selection: Automatic

Buffer number [0..31]:

0

Sample time:

-1

OK    Cancel    Help    Apply
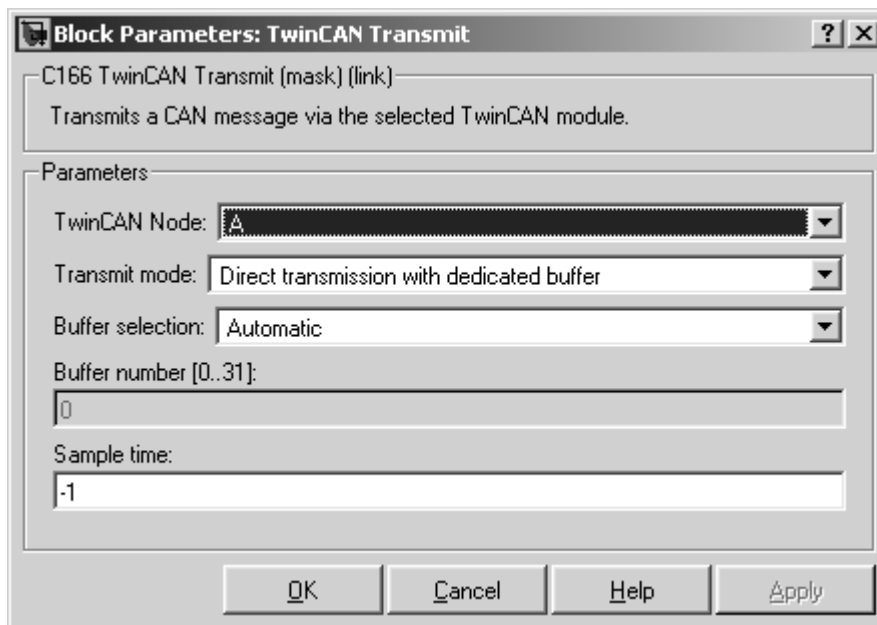
**TwinCAN Node**

Select node A or node B.

**Transmit mode**

Select one of the modes described above: direct transmission with dedicated buffer, or FIFO queue with dedicated buffer.

**Buffer selection**

Choose either automatic or manual selection of the hardware buffer number.

**Buffer number [0..31]**

This option is available only if the buffer selection is available and set to manual. You must select a buffer number between 0 and 31. Note if more than one message is ready to be transmitted, then the one in the lower buffer number will be sent first. Select buffer numbers such that the higher the message priority, the lower

the buffer number. Note that the hardware buffers are shared between node A and node B of the TwinCAN module.

**Sample time**

Choose -1 to inherit the sample time from the driving blocks. The TwinCAN Transmit block does not inherit constant sample times and runs at the base rate of the model if driven by invariant signals.

# Index