

# Embedded Target for Infineon C166<sup>®</sup> Microcontrollers

**For Use with Real-Time Workshop<sup>®</sup>**

- Modeling
- Simulation
- Implementation

User's Guide

*Version 1*



## How to Contact The MathWorks:



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



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



508-647-7000      Phone



508-647-7001      Fax



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

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

### *Embedded Target for Infineon C166 Microcontrollers*

© COPYRIGHT 2002–2005 by The MathWorks, Inc.

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

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered 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.

#### Revision History:

November 2002	Online only	Version 1.0 (Release 13+)
June 2004	Online only	Version 1.1 (Release 14)
October 2004	Online only	Version 1.1.1 (Release 14SP1)
March 2005	Online only	Version 1.2 (Release 14SP2)

## Getting Started

**1**

<b>What Is the Embedded Target for Infineon C166</b>	
<b>Microcontrollers?</b> .....	<b>1-3</b>
Feature Summary .....	<b>1-3</b>
<b>Prerequisites</b> .....	<b>1-5</b>
<b>Using This Guide</b> .....	<b>1-6</b>
<b>Installing the Embedded Target for Infineon C166</b>	
<b>Microcontrollers</b> .....	<b>1-8</b>
<b>Hardware and Software Requirements</b> .....	<b>1-9</b>
Host Platform .....	<b>1-9</b>
Hardware Requirements .....	<b>1-9</b>
Software Requirements .....	<b>1-10</b>
Switching Between Hardware Variants .....	<b>1-11</b>
Using Prebuilt RTW Libraries .....	<b>1-12</b>
<b>Setting Up and Verifying Your Installation</b> .....	<b>1-14</b>
Verifying MiniMon Settings .....	<b>1-14</b>
<b>Setting Up Your Target Hardware</b> .....	<b>1-16</b>
Jumper Settings for the phyCore-167 Development	
Board .....	<b>1-16</b>
<b>Setting Target Preferences</b> .....	<b>1-17</b>
<b>Creating a Make Variables Reference File for the Build</b>	
<b>Process</b> .....	<b>1-19</b>
Make Variables Reference File .....	<b>1-19</b>
Content of the Make Variables Reference File .....	<b>1-19</b>

Creating a New Make Variables Reference File Using the Tasking EDE .....	1-20
<b>Supported Blocks and Data Types .....</b>	<b>1-22</b>
<b>Overview of C166 Configuration Parameters .....</b>	<b>1-23</b>

## **Tutorial: Simple Example Applications for C166 Microcontrollers**

### **2**

<b>Introduction .....</b>	<b>2-2</b>
<b>Tutorial: Creating a New Application .....</b>	<b>2-3</b>
Before You Begin .....	2-3
Example Model 1: c166_serial_transmit .....	2-4
Generating and Downloading Code .....	2-6
Example 2: c166_serial_io .....	2-9
<b>Starting the Debugger on Completion of the Build Process .....</b>	<b>2-12</b>
Fixed-Point Example Model: c166_fuelsys .....	2-14
<b>Generating ASAP2 Files .....</b>	<b>2-16</b>

## **Integrating Your Own Device Drivers**

### **3**

<b>Integrating Hand-Coded Device Drivers with a Simulink Model .....</b>	<b>3-2</b>
<b>Preparing Input and Output Signals to the Device Driver Functions .....</b>	<b>3-3</b>

<b>Calling the Device Driver Functions from c166_main.c</b> .....	<b>3-6</b>
<b>Adding the I/O Driver Source to the List of Files to Build</b> .....	<b>3-8</b>
<b>Tutorial: Using the Example Driver Functions</b> .....	<b>3-10</b>

## **Custom Storage Class for C166 Microcontroller Bit-Addressable Memory**

### **4**

<b>Specifying C166 Microcontroller Bit-Addressable Memory</b> .....	<b>4-2</b>
<b>Using the Bitfield Example Model</b> .....	<b>4-3</b>

## **Execution Profiling**

### **5**

<b>Overview of Execution Profiling</b> .....	<b>5-2</b>
Definitions .....	<b>5-3</b>
Execution Profiling Blocks .....	<b>5-3</b>
<b>Real-Time Workshop Options for Execution</b>	
<b>Profiling</b> .....	<b>5-4</b>
Execution Profiling .....	<b>5-4</b>
Number of Data Points .....	<b>5-5</b>
Task Scheduler Overrun Options .....	<b>5-5</b>
<b>Multitasking Demo Model</b> .....	<b>5-7</b>
Running the Multitasking Demo .....	<b>5-7</b>
Interpreting the MATLAB Graphic .....	<b>5-9</b>

## Blocks — Categorical List

### 6

<b>Embedded Target for Infineon C166 Microcontrollers</b>	
<b>Block Library</b> .....	<b>6-2</b>
<b>C166 Drivers Library</b> .....	<b>6-3</b>
Configuration Class Blocks .....	<b>6-6</b>
Using Block Reference Pages .....	<b>6-7</b>

## Blocks — Alphabetical List

### 7

## Index

# Getting Started

---

This section contains the following topics:

“What Is the Embedded Target for Infineon C166 Microcontrollers?”  
(p. 1-3)

Overview of the product and the use of the Embedded Target for Infineon C166<sup>®</sup> Microcontrollers in the development process.

“Prerequisites” (p. 1-5)

What you need to know before using the Embedded Target for Infineon C166 Microcontrollers.

“Using This Guide” (p. 1-6)

Suggested path through this document to get you up and running quickly with the Embedded Target for Infineon C166 Microcontrollers.

“Installing the Embedded Target for Infineon C166 Microcontrollers”  
(p. 1-8)

Installation of the product.

“Hardware and Software Requirements” (p. 1-9)

Hardware platforms supported by the product; development tools (e.g., compilers, debuggers) required for use with the product.

“Setting Up and Verifying Your Installation” (p. 1-14)

Overview of setting up your development tools and hardware to work with the Embedded Target for Infineon C166 Microcontrollers, and verifying correct operation.

“Setting Up Your Target Hardware”  
(p. 1-16)

Port connections and jumper settings.

- “Setting Target Preferences” (p. 1-17) Configuring environmental settings and preferences associated with the Embedded Target for Infineon C166 Microcontrollers.
- “Creating a Make Variables Reference File for the Build Process” (p. 1-19) This section explains the purpose of the Make Variables Reference File file specified in the C166 Target Preferences. You will need to understand these and generate new files if you want to change the default settings supplied with the Embedded Target for Infineon C166 Microcontrollers.
- “Supported Blocks and Data Types” (p. 1-22) Requirements and restrictions.
- “Overview of C166 Configuration Parameters” (p. 1-23) Links to information about C166 Options in the Configuration Parameters dialog.



## What Is the Embedded Target for Infineon C166 Microcontrollers?

The Embedded Target for Infineon C166<sup>®</sup> Microcontrollers is an add-on product for use with the Real-Time Workshop<sup>®</sup> 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<sup>®</sup>, Stateflow<sup>®</sup>, and the Real-Time Workshop Embedded Coder, the Embedded Target for Infineon C166 Microcontrollers lets you

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

### Feature Summary

- Automatic generation of 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

- 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

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.

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

In addition, if you want to understand and use the device driver blocks in the 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 1-14.
- 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 through the example provided in “Tutorial: Using the Example Driver Functions” on page 3-10.
- 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 — Categorical List” 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**, or at the command line enter

```
demo simulink 'Embedded Target for Infineon C166  
Microcontrollers'
```

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, NT, 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:

- Phytex phyCORE-167 ST10F269
- Phytex phyCORE-167 C167CS
- Phytex kitCON-167 C167CR
- Infineon XC167I 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 1-11.

This guide assumes that you are working with the Phytex phyCORE-167CS development board, and documents specific settings and procedures for use with the Phytex 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

Simulink Fixed Point is strongly recommended but not essential; it is required 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/](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. The Embedded Target for Infineon C166 Microcontrollers currently supports the cross-development tools listed below:

- Tasking C Cross-Compiler and debugger toolchain (version 8.0 r2)

---

**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 1-14.

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

- Phytex phyCORE-C167CS (RAM)
- Phytex phyCORE-C167CS (flash)
- Infineon XC167CI Starter Kit
- Phytex phyCORE-ST10F269
- Phytex kitCON-C167CR (RAM)
- Phytex kitCON-C167CR (flash)

If your hardware variant is not on this list, you will need to change the default configuration settings. See “Creating a Make Variables Reference File for the Build Process” on page 1-19.

When switching between target configurations, you should review all of your Target Preferences and ensure that they are set appropriately for the new configuration. It is necessary to change the Target Preferences only once and the new settings will take effect for all subsequent builds.

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.

## Using Prebuilt RTW Libraries

You may need to use this option when switching between hardware variants. The option to **Use prebuilt RTW libraries** is found in the **C166 Options** (under **Real-Time Workshop** in the Configuration Parameters dialog). Checking this option causes the build process to link against a prebuilt static library that contains object code for Real-Time Workshop library functions. If this option is not checked, then all Real-Time Workshop library functions are recompiled in the model build area; this can be a lengthy process when the model is built for the first time.

The prebuilt static library `matlabroot\rtw\c\lib\win32\rtwlib_c166.lib` has been built with options selected for C167CS. For other processor variants, it may be necessary to rebuild this library.

To do this, open a command prompt and run the following commands:

```
cd matlabroot/toolbox/rtw/targets/c166/libs/rtwlib
set C166ROOT=<path to c166 compiler>
set MATLABROOT=<path to matlab root directory>
set CFLAGS=<required C compiler flags>
matlabroot\rtw\bin\win32\gmake -f makefile clean
matlabroot\rtw\bin\win32\gmake -f makefile
```

Note that you can obtain appropriate settings for CFLAGS by inspecting the `OPT_CC` variable in the `MakeVariablesReferenceFile` that is specified in the C166 Target Preferences Setup. See “Creating a Make Variables Reference File for the Build Process” on page 1-19.

When the make command is complete, the output from the command window should indicate that all the required Real-Time Workshop library files are being compiled. This may take a few minutes.

To build the library as originally supplied simply repeat the above steps except with

```
set CFLAGS=
```

i.e. ensure that CFLAGS is undefined. In this case the library will use the default flags (for C167CS) that are specified in the makefile.

### **Spaces in Path**

Note that if either MATLAB or the compiler are installed in a directory with spaces in the pathname, you must make sure that the environment variables C166ROOT and MATLAB\_ROOT are specified in 8dot3 format. You can run the command `dir /x` from the Windows command prompt to find out the 8dot3 formatted directory names.

## 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.de/>

From here select **Site Map > Products > 16-bit Microcontrollers**, then locate ‘minimon\_c16x\_223\_setup\_2.zip’.

Be sure to install the 2.2.3 version. Earlier versions do not contain all the required controller configurations. Once installed 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 illustration.

This configuration has been verified with both a phyCORE C167CS board and a Phytex kc167 (C167CR). In general, you should choose configuration settings that are consistent with the values specified in the Tasking EDE project that is used to create your Make Variables Reference File.

The screenshot shows the 'Configure' dialog box with the following settings:

- Initialize register:**
  - SYSCON: 0085 h
  - BUSCON0: [ ] h
  - ADDRSEL1: 0006 h
  - BUSCON1: 049F h
  - ADDRSEL2: [ ] h
  - BUSCON2: [ ] h
  - ADDRSEL3: [ ] h
  - BUSCON3: [ ] h
  - ADDRSEL4: [ ] h
  - BUSCON4: [ ] h
  - Generic 1: [ ] h, Addr: [ ] h
  - Generic 2: [ ] h, Addr: [ ] h
- Controller type:** C167CR, Clk rate: 20000000 Hz
- Memory:**

A	M		
X		SFR	: 00FE00-00FFFF
X		ESFR	: 00F000-00F1FF
X		IRAM	: 00F600-00FDFF
		XRAM	: 00E000-00E7FF
		CAN	: 00EF00-00EFFF
- Initial command calls:**
  - EINIT

Buttons: Clear, Cancel, OK

## 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.

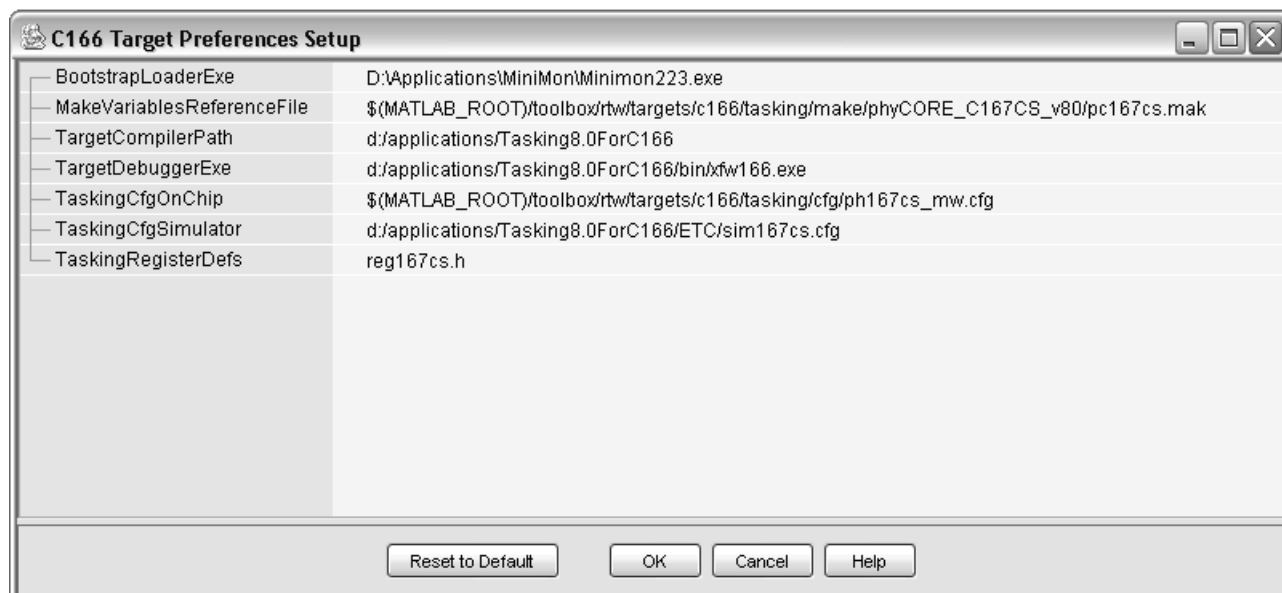
### Jumper Settings for the phyCore-167 Development Board

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



- 2 Edit the settings for your cross-development environment:
  - BootstrapLoaderExe specifies the path to your download utility (MiniMon).
  - MakeVariablesReferenceFile specifies a makefile that is used as a reference for building applications created with Embedded Target for Infineon C166 Microcontrollers. For further details on creating and using this file, see “Make Variables Reference File” on page 1-19.

- `TargetCompilerPath` specifies the path to your compiler (Tasking).
- `TargetDebuggerExe` specifies the path to your debugger executable (CrossView)
- `TaskingCfgOnChip` specifies the name of a CrossView configuration file that will be used to start CrossView when the build action is set to `Download_and_run_with_debugger`. Consult the CrossView documentation, C166/ST10 CrossView Pro Debugger User's Guide for further details.
- `TaskingCfgSimulator` specifies the name of a CrossView configuration file that will be used to start CrossView when the build action is set to `Run_with_simulator`.
- `TaskingRegisterDefs` specifies an include file that may be used in the automatically generated C source code; this file should be located in the `include` subdirectory of your Tasking compiler installation. This target preference allows you to select a register definitions file that is appropriate for your target hardware and consistent with the settings in the Make Variables Reference File. The file contains definitions of all the special function registers, etc., that may be dependent upon your target hardware. See the Tasking documentation, C166/ST10 C Cross-Compiler User's Guide, for further details.

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

See the next section, "Creating a Make Variables Reference File for the Build Process" on page 1-19, for more information on using the configuration files specified in the target preferences.



## Creating a Make Variables Reference File for the Build Process

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

You can change the default settings by supplying your own configuration files in the C166 Target Preferences Setup dialog. The following information explains the purpose of the Make Variables Reference File and how to create a new one for different memory models or hardware variants.

### Make Variables Reference File

The target preference `MakeVariablesReferenceFile` contains make variables that are copied and used by the Embedded Target when building a model. The Make Variables Reference File contains information that is specific to the C166 hardware variant as well as build settings such as the memory configuration.

The Make Variables Reference Files provided with Embedded Target for Infineon C166 Microcontrollers allows you to build an application for selected evaluation boards such as the Phytex phyCORE-C167CS.

If none of the supplied Make Variables Reference Files are suitable for your hardware configuration, you must create a new one. The easiest way to do this is by creating a new project using the Tasking EDE, as described below.

### Content of the Make Variables Reference File

The target preference `MakeVariablesReferenceFile` allows a makefile to be specified that is used as a reference for obtaining build configuration settings. The specified makefile is not used directly, rather it is examined and used to provide the following information:

- `OPT_CC` is a make variable containing flags used during the compile stage of the build process.

- `OPT_MPP` is a make variable containing flags used during the macro preprocessor stage of the build process.
- `OPT_LC` is a make variable containing flags used during the link/locate stage of the build process.
- `start.asm` is the startup code that will be copied to the build directory then compiled and linked with the rest of the application.
- `Filename.ilo` is a file referenced within `OPT_LCC` that contains extra instructions for the linker/locator. The content of this file is copied to the build directory.

## Creating a New Make Variables Reference File Using the Tasking EDE

To create a new Tasking EDE project along with the required Make Variables Reference File, follow these steps:

**1** Create a new folder for the project.

**2** Copy source files for a sample application, e.g.,

```
matlabroot\toolbox\rtw\targets\c166\tasking\make\phyCORE_C167CS_v80\main.c
```

to this new directory.

**3** Open the Tasking EDE.

**4** Right-click the root of the project tree and select **Add New Project**, and create a new project in the folder created for that purpose.

**5** Click the + icon to scan the source file into the new project.

**6** In the project tree, right-click the new project and set it as the current project.

**7** Under the menu item **Project > Project Options**, configure the required settings for your hardware environment. This should include the following:

- a** Set the CPU type (and allow EDE to set registers accordingly).
- b** In the Linker/Locator section, check the option to generate Intel Hex files.

- c** In the CrossView Pro/Execution Environment, select the required evaluation board and confirm that startup registers should be set to default values for this execution environment; this should ensure that X-Bus peripherals such as CAN are enabled on hardware where they are available.
- 8** Right-click on the project and select **Build**. Check that the project builds successfully and that a new Make Variables Reference File with `.mak` file extension has been created in the project directory.
- 9** It is a good idea to test that you can run the application `main.c` on your hardware; to do this you should run CrossView and download the compiled application. You may wish to provide your own sample application instead of `main.c`, or make a modification to `main.c` to test specific features of your hardware.
- 10** The new `.mak` file is now ready for use with Embedded Target for Infineon C166 Microcontrollers; to use it, you must specify the path to this new file in your target preferences.

Consult the Tasking documentation C166/ST10 C Cross-Compiler User's Guide and C166/ST10 Cross-Assembler, Linker/Locator, Utilities User's Guide for further details.

Tasking EDE is the best place to start if you want to configure the startup code, but you can also try using the Infineon Digital Application Engineer DAve. The freeware DAve is also useful for developing device drivers. See "Integrating Hand-Coded Device Drivers with a Simulink Model" on page 3-2.

## 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 3-10.

## Overview of C166 Configuration Parameters

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

### Build action

#### Download\_and\_run

This option is described using the C166\_serial\_transmit model in “Tutorial: Creating a New Application” on page 2-3.

#### Download\_and\_run\_with\_debugger and Run\_with\_simulator

These options are described in “Starting the Debugger on Completion of the Build Process” on page 2-12. The CrossView configuration files for these two options are specified in the Target Preferences: TaskingCfgOnChip and TaskingCfgSimulator, see “Setting Target Preferences” on page 1-17.

### CrossView startup options file

This field is available when one of the options Run\_with\_simulator or Download\_and\_run\_with\_debugger is selected for the **Build action**. By default, the field is empty; in this case a file containing start-up options for CrossView is created automatically and is used to run CrossView at the end of the build process. By specifying a file containing CrossView start-up options the default file is overridden with your own start-up options. See the section "Startup Options" in the CrossView user manual for details on available startup options.

### 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.

Execution profiling options:

**Maximum number of concurrent base-rate overruns**

**Maximum number of concurrent sub-rate overruns**

**Execution profiling**

**Number of data points**

These are options for task execution profiling. See “Overview of Execution Profiling” on page 5-2.

**Use prebuilt RTW libraries**

This option causes the build process to link against a prebuilt static library that contains object code for Real-Time Workshop library functions. This can save time when your processor variant can use the prebuilt library. See “Using Prebuilt RTW Libraries” on page 1-12.

# Tutorial: Simple Example Applications for C166 Microcontrollers

---

This section includes the following topics:

“Introduction” (p. 2-2)

An overview of the Embedded Target for Infineon C166 Microcontrollers real-time target, other components required to generate stand-alone real-time applications, and the process of deploying generated code on target hardware.

“Tutorial: Creating a New Application” (p. 2-3)

A hands-on exercise in building two simple applications from demo models, including downloading and executing generated code on a target board.

“Starting the Debugger on Completion of the Build Process” (p. 2-12)

This exercise shows you how to generate code and commence debugging automatically as part of the build process. Depending on your debugger, you can debug the application either on-chip or on a hardware simulator.

“Generating ASAP2 Files” (p. 2-16)

How to generate ASAP2 files for your models.

# Introduction

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

- The Embedded Target for Infineon C166 Microcontrollers real-time target
- The example models provided: `c166_serial_transmit.mdl` and `c166_serial_io.mdl`
- The Tasking 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 applications. You do not need to hand-write any C code to integrate the generated code into a final application.

The tutorial “Tutorial: Creating a New Application” on page 2-3 uses two blocks from the Embedded Target for Infineon C166 Microcontrollers library. For complete information on the Embedded Target for Infineon C166 Microcontrollers library blocks, see Chapter 6, “Blocks — Categorical List”



## 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:

- Phytex 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.

---

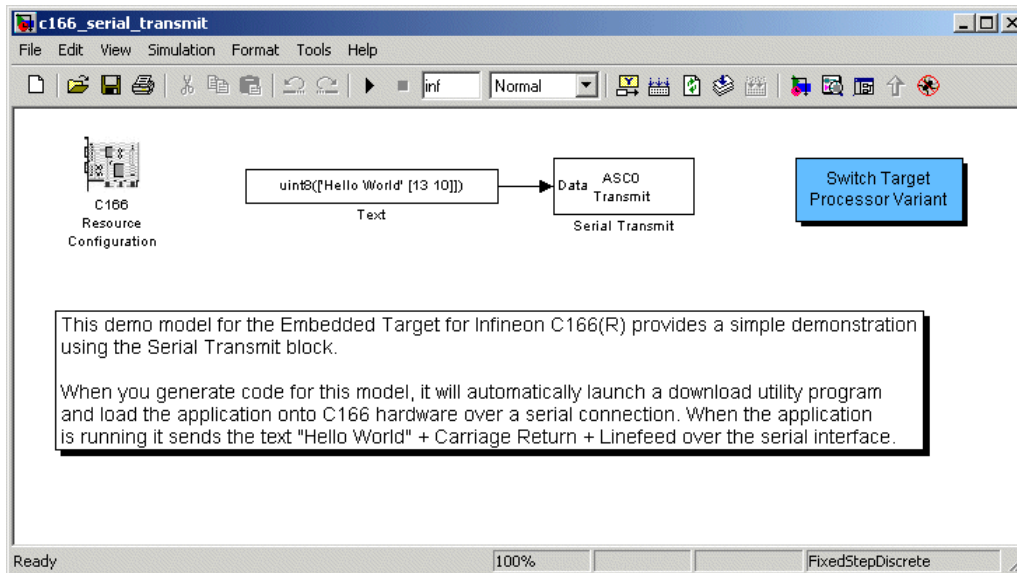
### Example Model 1: c166\_serial\_transmit

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.

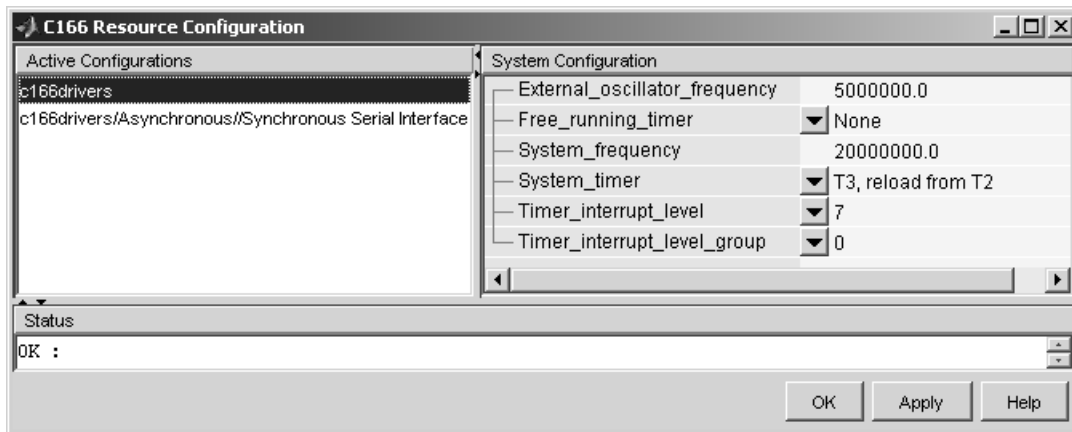
The purpose of the C166 Resource Configuration object is to provide information to other blocks in the model. Unlike conventional blocks, the C166 Resource Configuration object is not connected to other blocks via input or output ports. Instead, driver blocks (such as the ASC0

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 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.

---

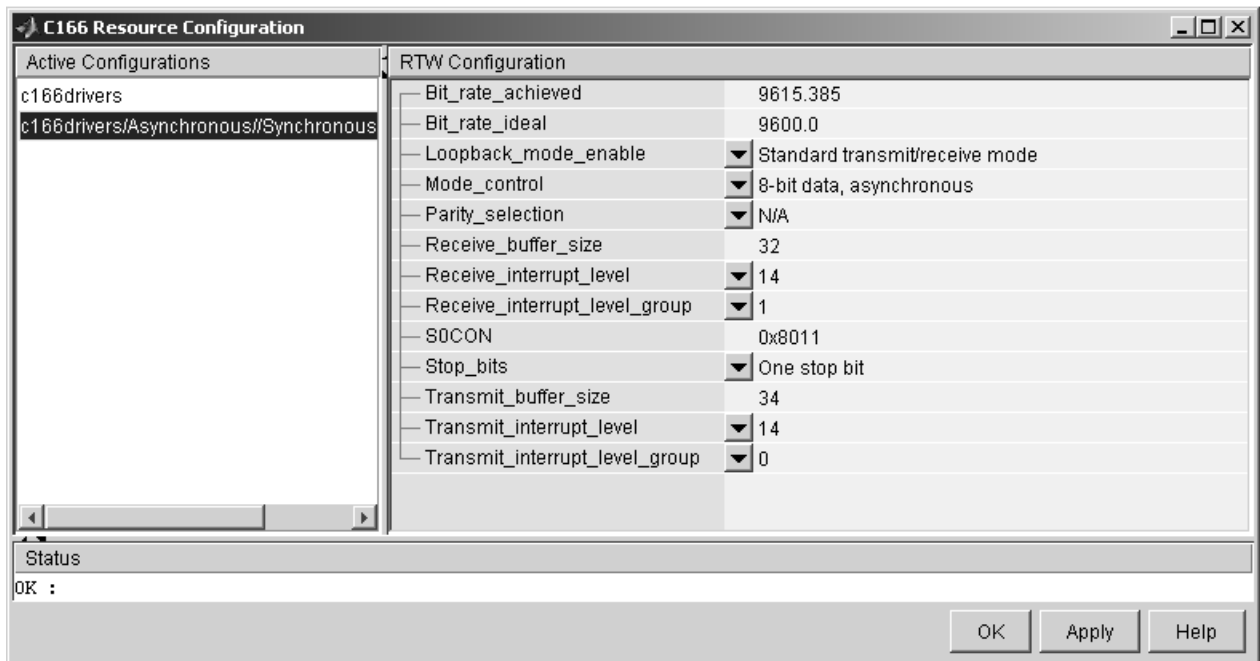
**Note** If hardware is running at a system frequency other than 20 MHz, you must change this parameter appropriately.

---

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.

- 3 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 **RTW Configuration** pane on the right. Do not edit any of these parameters for this tutorial. To learn more about the C166 Resource Configuration object, see C166 Resource Configuration.

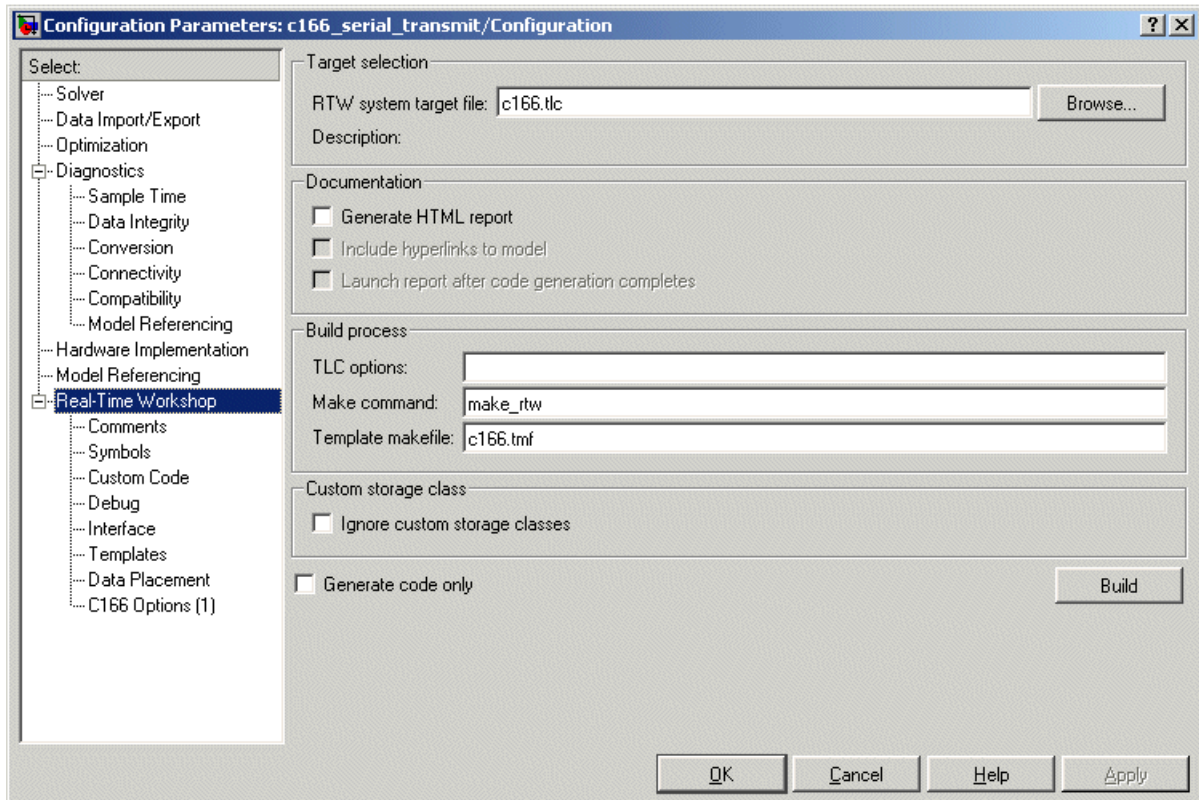
- 4 Close the **C166 Resource Configuration** window before proceeding.

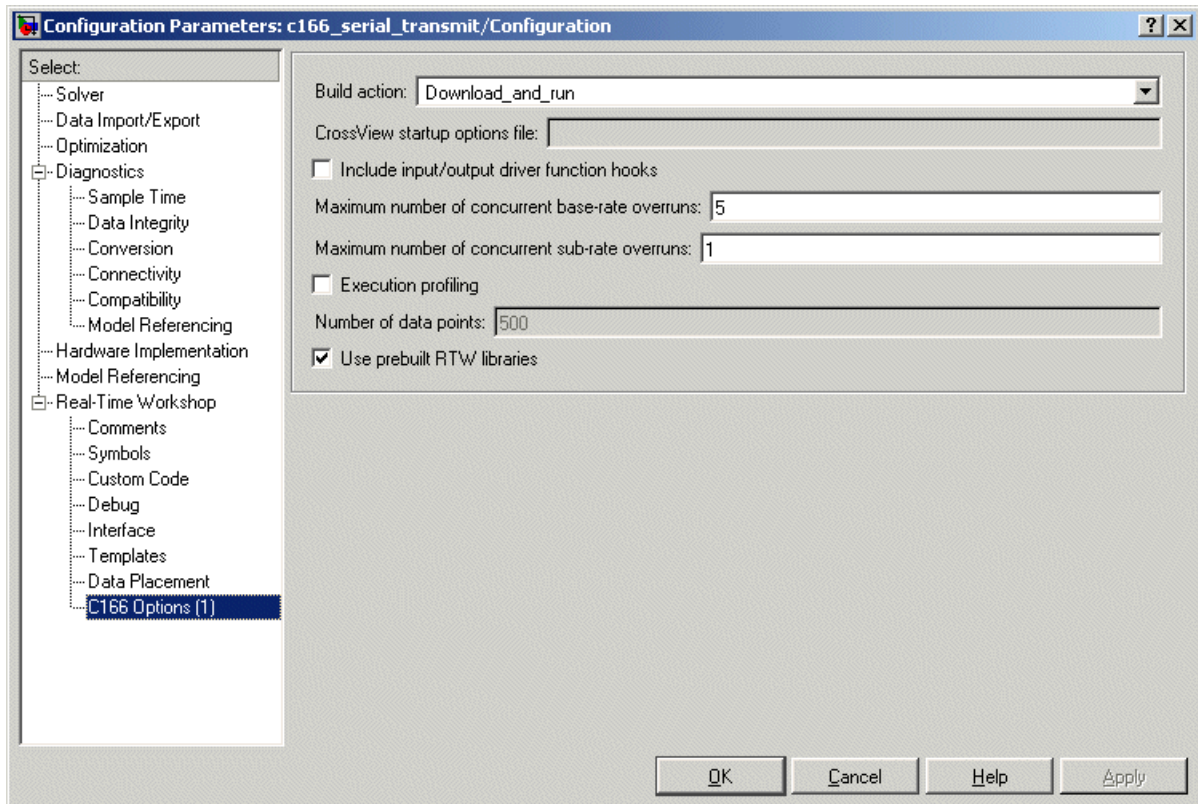
## Generating and Downloading Code

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 Select C166 Options (1) (under Real-Time Workshop in the tree).**



Make sure that the **Build action** is `Download_and_run`. When you generate code for this model, it automatically starts a download utility program and loads the application onto the C166 microcontroller hardware over a serial connection. The code then begins execution on the target.

- 4 Return to the Real-Time Workshop options (click **Real-Time Workshop** in the tree) and click **Build**.

Note that you could have gone 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. When MiniMon is started, a 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.

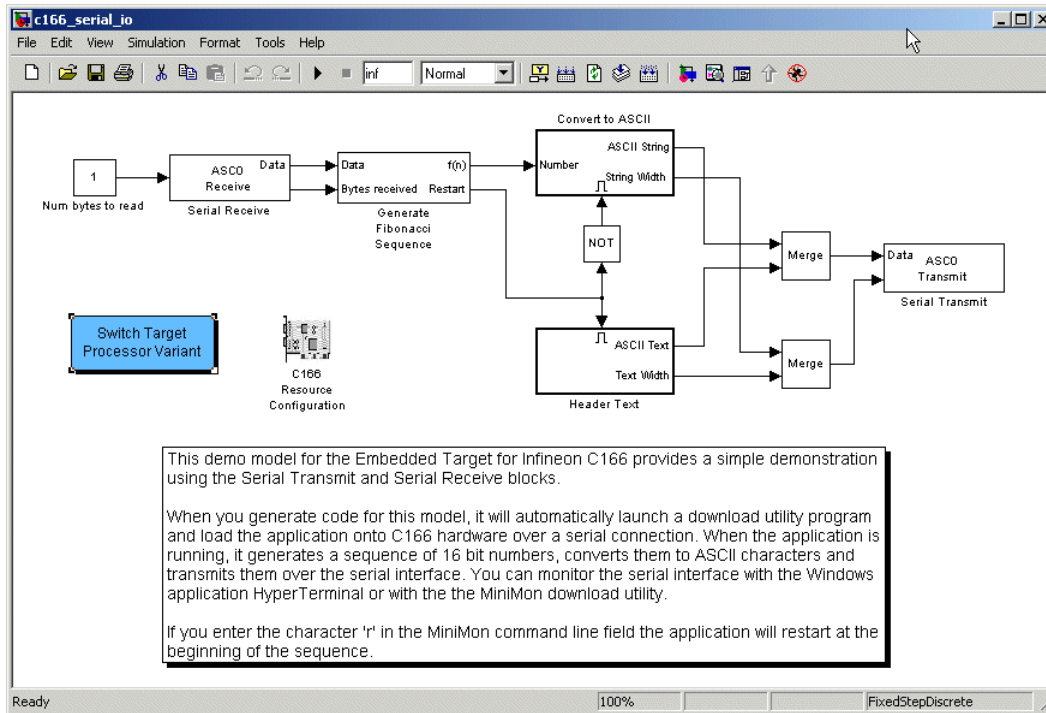
## Verifying Code Execution on the Target

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

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

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

You can see progress messages in the MiniMon window as it connects and then downloads to the target. MiniMon 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.

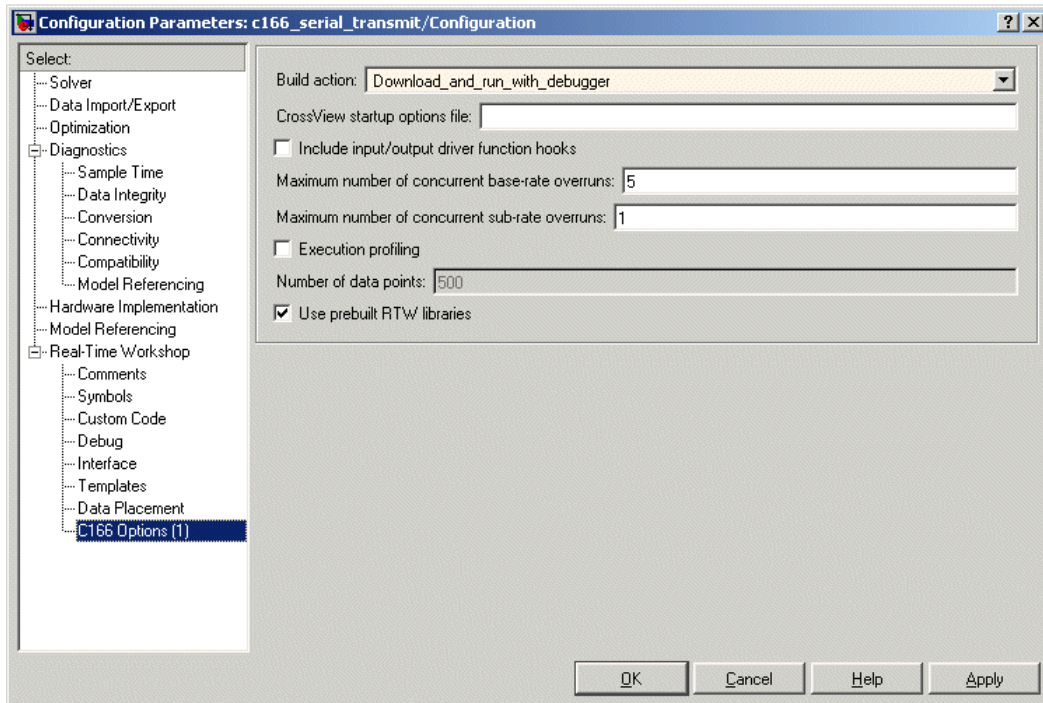
### Starting the Debugger on Completion of the Build Process

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.mdl`. This model is designed to show you how to integrate your own hand-coded device drivers with automatically generated code using Embedded Target for Infineon C166 Microcontrollers. This model is covered in detail in Chapter 3, “Integrating Your Own Device Drivers”. You 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.mdl`.
- 2 Select **Simulation > Configuration Parameters**.
- 3 Select **C166 Options (1)** (under **Real-Time Workshop** in the tree).



- 4 Select the **Build action** `Run_with_simulator` or `Download_and_run_with_debugger`.
- 5 Before generating code, check that your target preferences related to the debugger are correctly configured. See “Setting Target Preferences” on page 1-17.
- 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. Attempting to use the debugger in this case causes an error.

---

See also the next section, “Fixed-Point Example Model: c166\_fuelsys” on page 2-14, for an example demo that starts the debugger in simulation mode rather than on-chip.

### **Fixed-Point Example Model: c166\_fuelsys**

The c166\_fuelsys model is derived from the demo fuelsys.mdl. 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. This demo starts the debugger in simulation mode rather than on-chip.

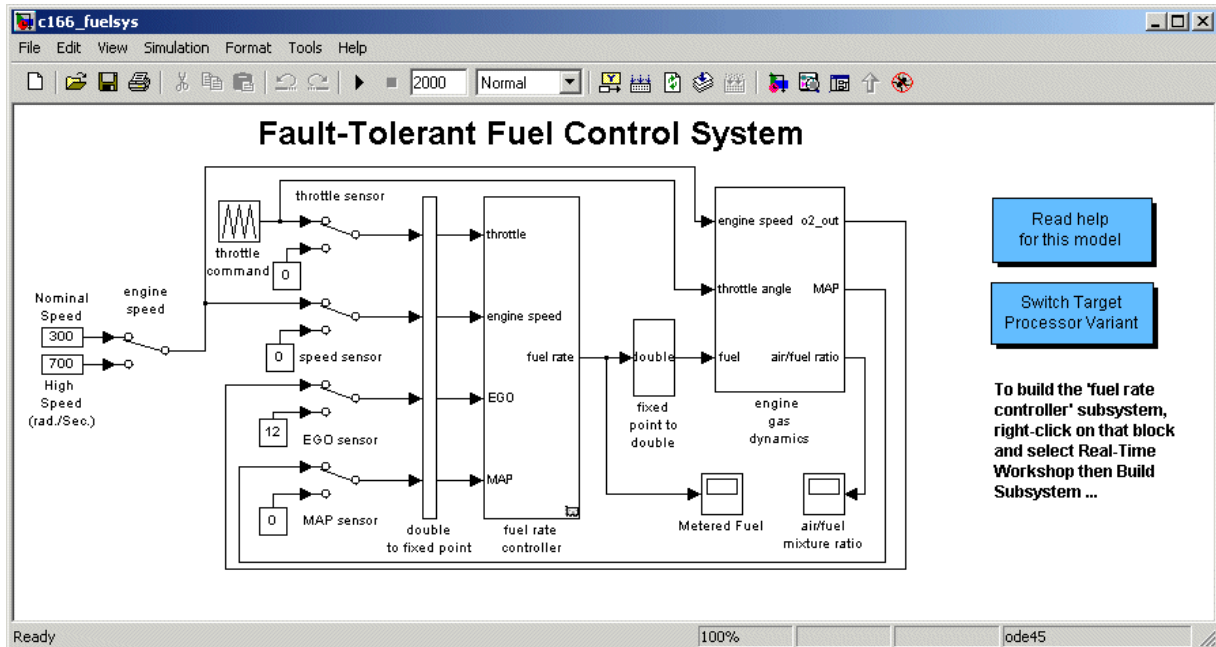
---

**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.mdl.



2 Right-click the fuel rate controller block.

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

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

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

## 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.

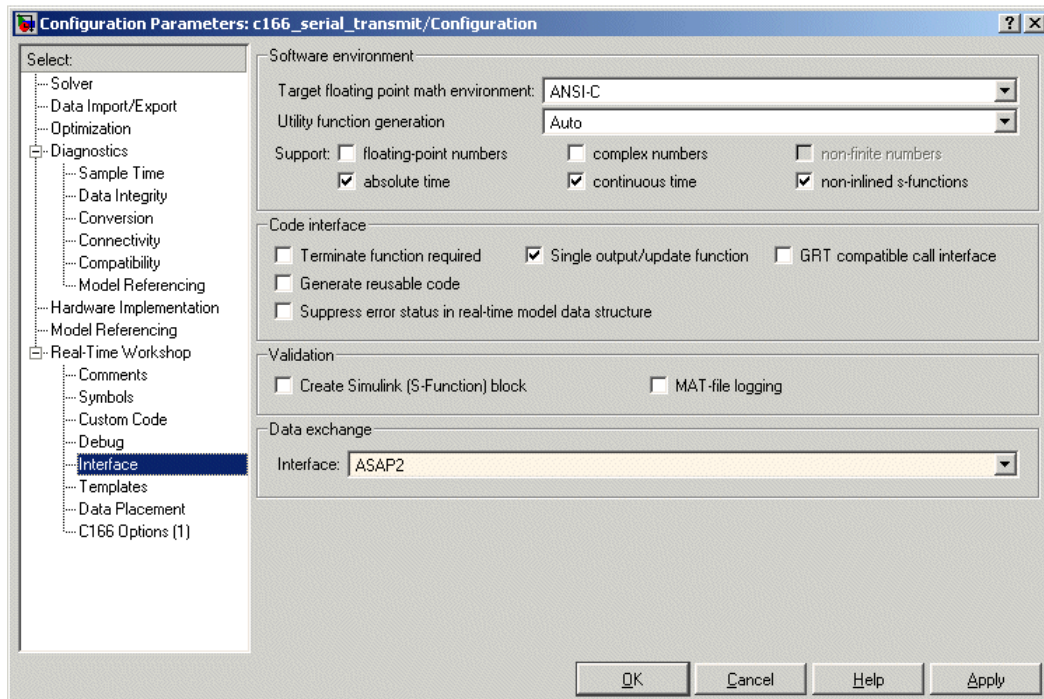
Before you begin generating ASAP2 files with the Embedded Target for Infineon C166 Microcontrollers, you should read the "Generating ASAP2 Files" section of the Real-Time Workshop documentation. That section describes how to define the signal and parameter information required by the ASAP2 file generation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

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

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.mdl`.



# 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 <code>c166_user_io.mdl</code> as an example.
“Calling the Device Driver Functions from <code>c166_main.c</code> ” (p. 3-6)	Real-Time Workshop settings to call your hand-coded device driver functions.
“Adding the I/O Driver Source to the List of Files to Build” (p. 3-8)	How to customize the Real-Time Workshop <code>make</code> command to integrate your device driver code.
“Tutorial: Using the Example Driver Functions” (p. 3-10)	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 <code>c166_user_io.mdl</code> 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.mdl`.

An alternative approach is to create Simulink I/O blocks that automatically generate the device driver code. This approach may be worth considering if you 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.de/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.mdl`. 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.

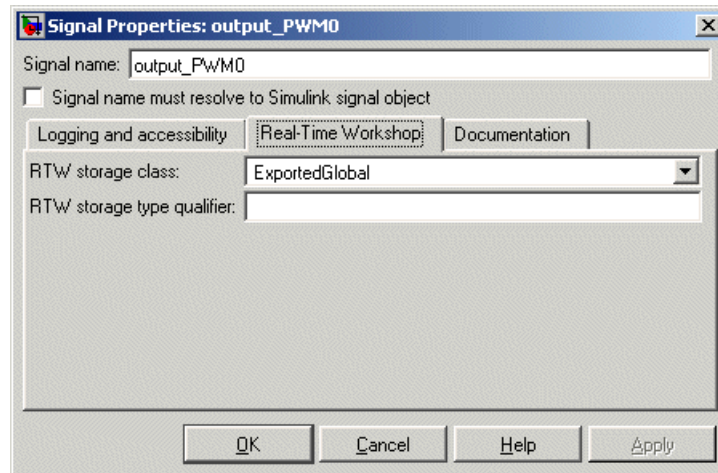
When structuring your model in this way, you should make sure that all the input and output signals to the control algorithm are channeled through top-level input or output ports in the control algorithm subsystem.

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.mdl`.
- 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.mdl` uses a custom storage class, which uses a single bit in a bitfield variable. See “Tutorial: Using the Example Driver Functions” on page 3-10 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.

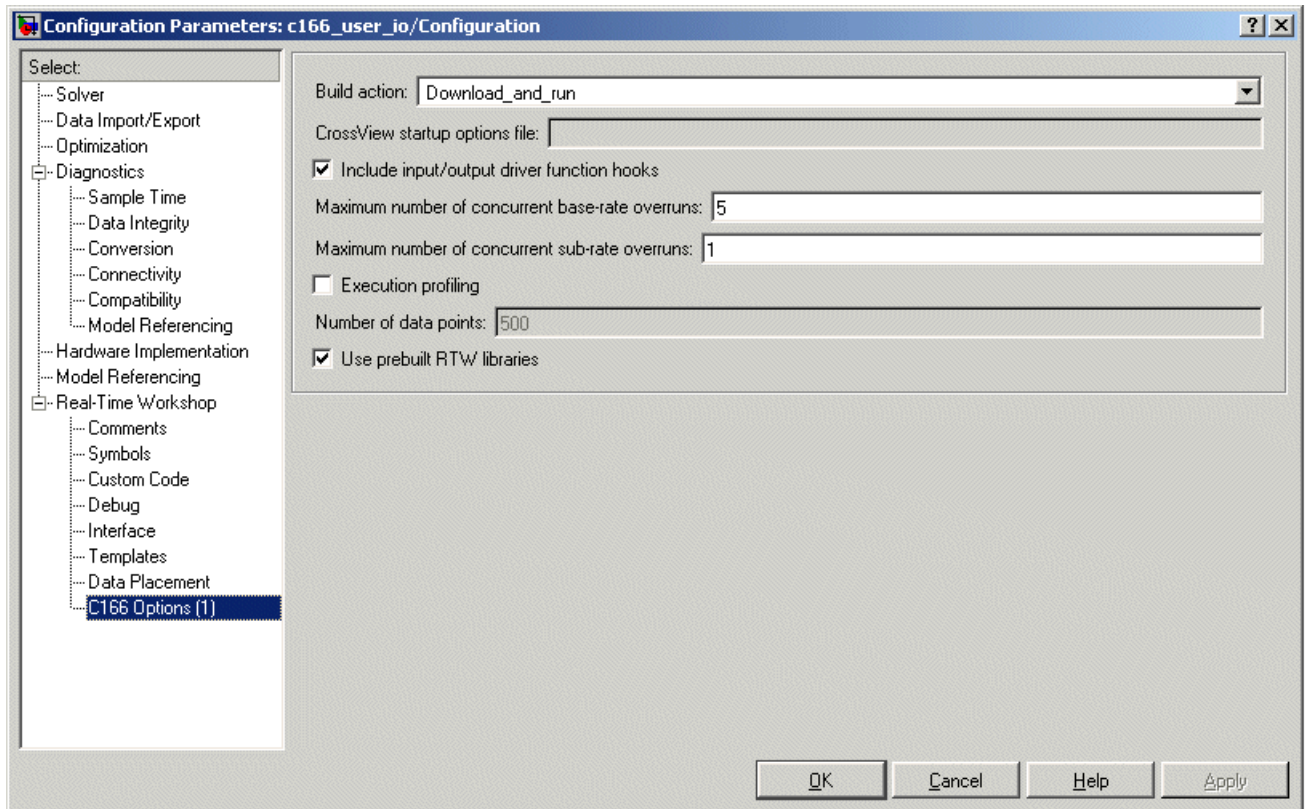
## Calling the Device Driver Functions from c166\_main.c

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  $i=1, 2, \dots$

`sub_rate_i_model_outputs` — write model outputs, called at the start of sub-rate  $i$ , where  $i=1, 2, \dots$

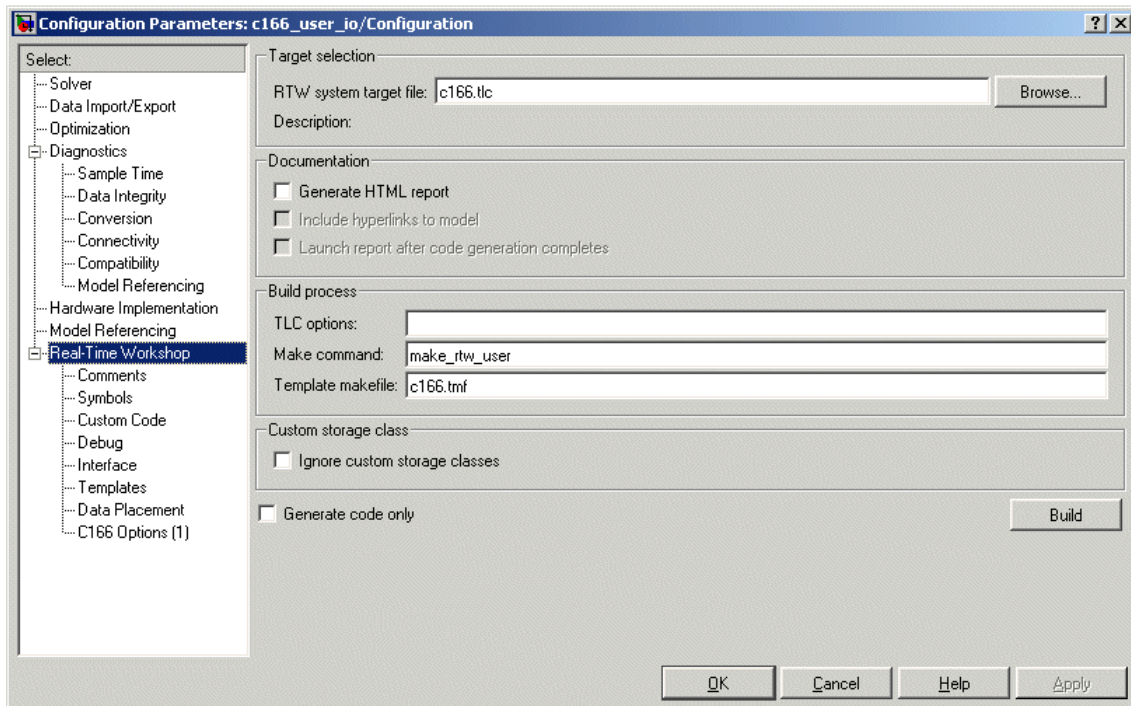
If you are using the automatically generated `c166_main.c`, then these function names are fixed.

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 3-10.

## 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. To do this, you must add some extra arguments to the `make_rtw` command in the **Real-Time Workshop** tab of the Simulation Parameters dialog:

- 1 Select **Real-Time Workshop** in the tree.



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

You must specify the names of the additional source files, e.g.,

```
make_rtw USER_SRCS = "file1.c file2.c" USER_INCLUDES =
"-Iincludedir1 -Iincludedir2"
```



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

```
make_rtw_user.m
```

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

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

You can examine an example of this custom make command in the example model `c166_user_io`. See the instructions in “Tutorial: Using the Example Driver Functions” on page 3-10. 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`

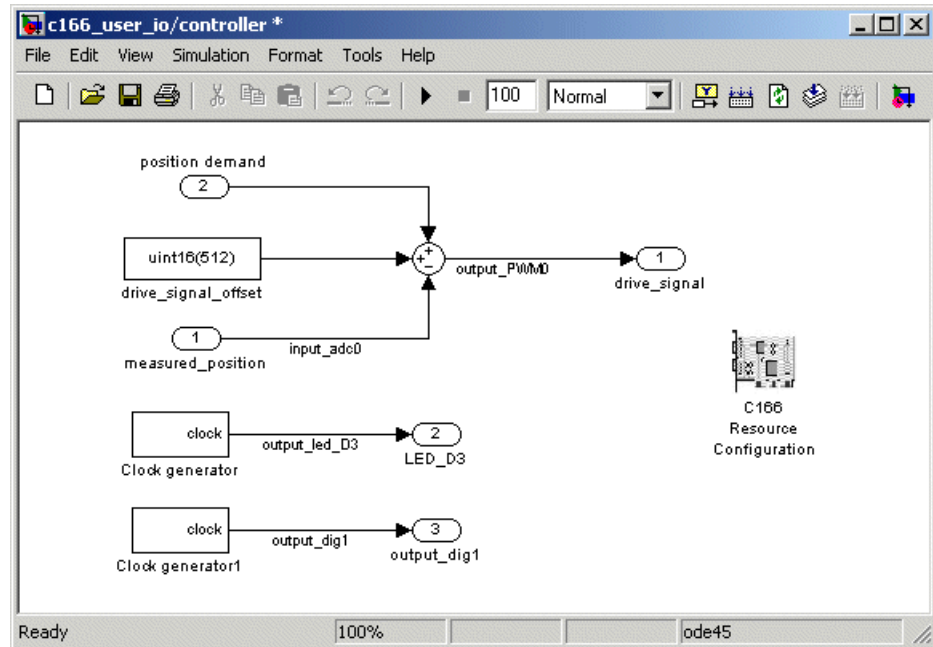
For `input_adc0`, the variable is defined in the hand code and referenced in the Real-Time Workshop code.

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:

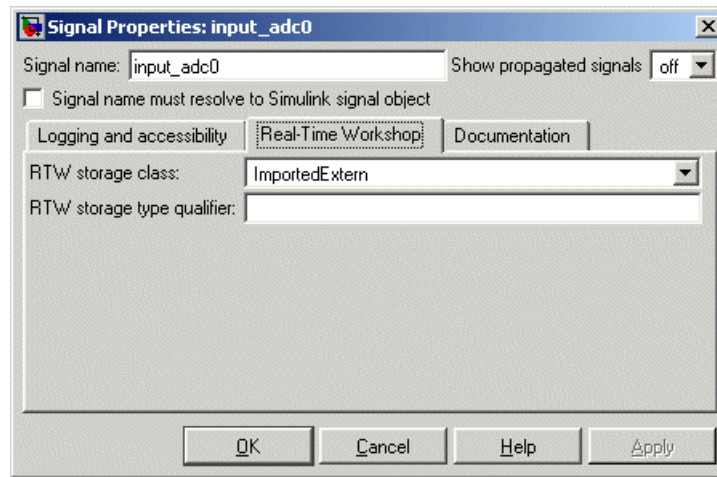
- 1 Open the model `c166_user_io.mdl`.





### 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`.

```

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 /*-----*

```

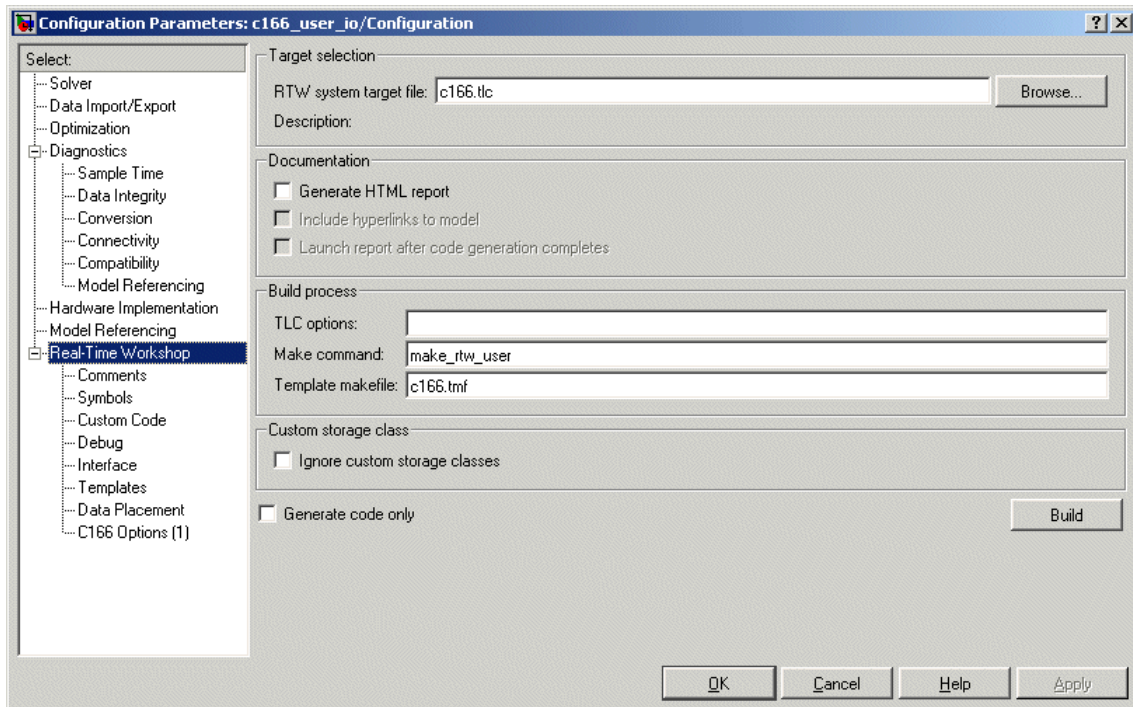
- 5 Click the `user_io.h` tab, as shown above. Here you can see `extern uint16_T input_adc0` under the heading `Declare variables that are imported by the model`. Also look at the `#include` directive in `user_io.c`.

The extern declaration and incorporating the header file into the build makes it possible for the device driver functions to read or write this variable that is defined in the Real-Time Workshop generated code.

**6** In the controller subsystem, select **Simulation > Configuration Parameters**. The Configuration Parameters dialog opens.

**7** Select **Real-Time Workshop** in the tree, and in the **Build process** pane look at the **Make command**:

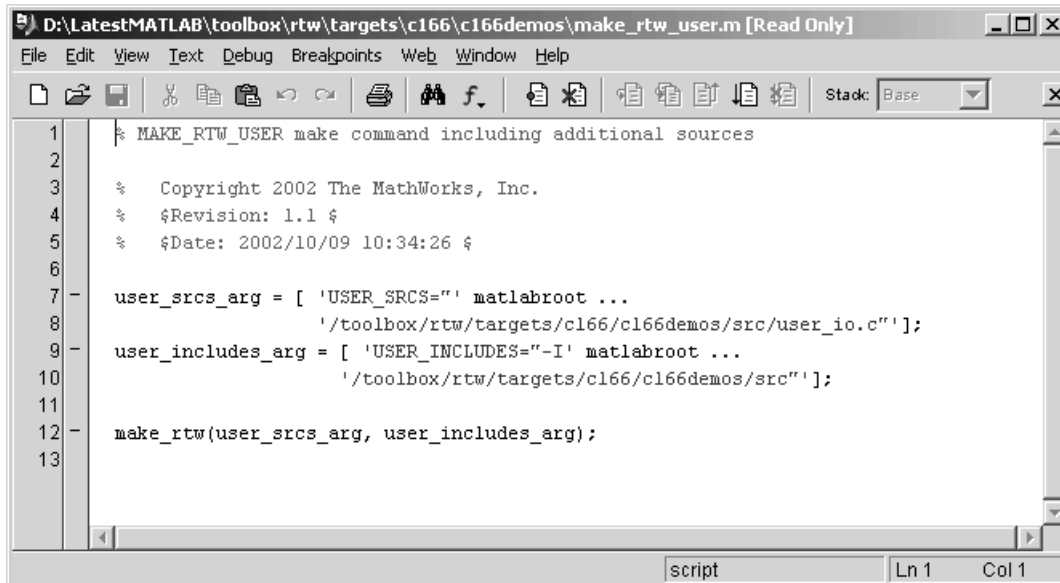
```
make_rtw_user
```



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

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

```
edit make_rtw_user
```



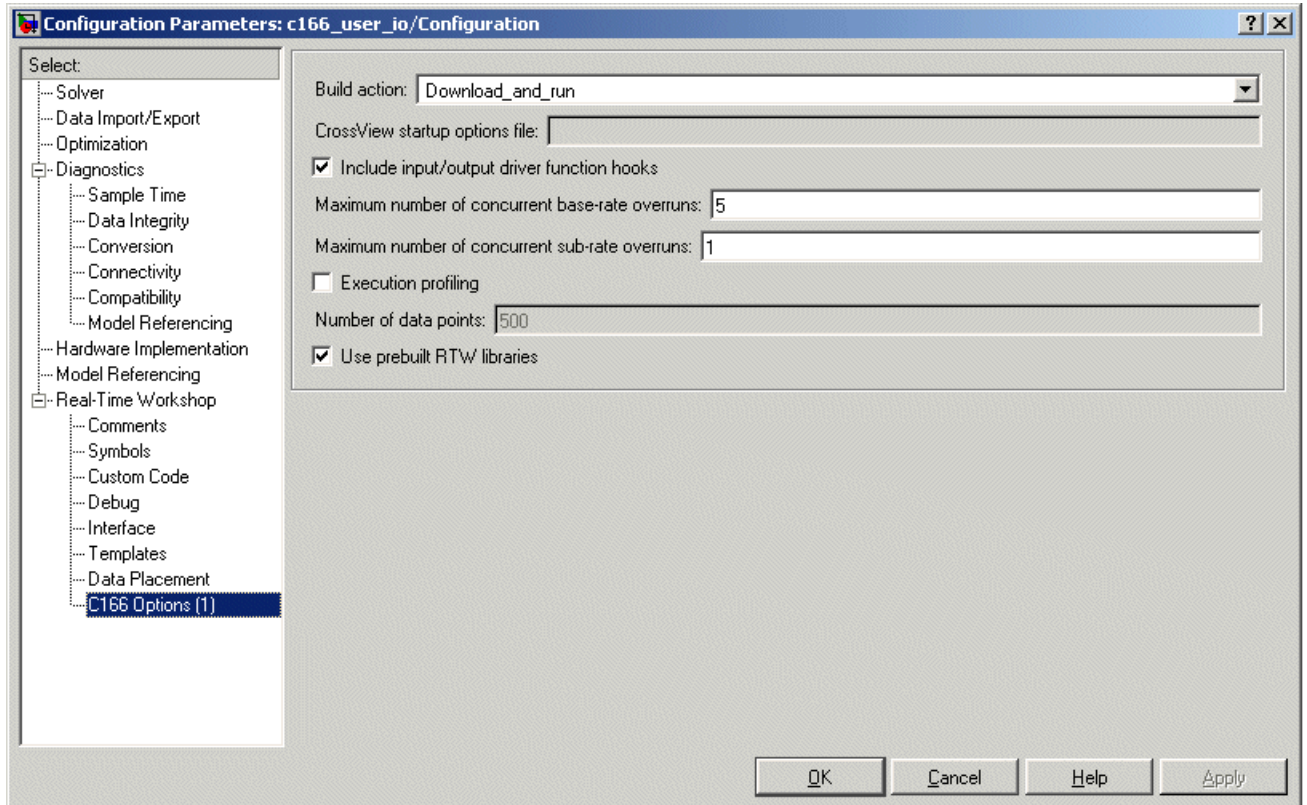
```

1  % MAKE_RTW_USER make command including additional sources
2
3  % Copyright 2002 The MathWorks, Inc.
4  % $Revision: 1.1 $
5  % $Date: 2002/10/09 10:34:26 $
6
7  user_srcs_arg = [ 'USER_SRCS="' matlabroot ...
8                  '/toolbox/rtw/targets/c166/c166demos/src/user_io.c"'];
9  user_includes_arg = [ 'USER_INCLUDES="-I' matlabroot ...
10                      '/toolbox/rtw/targets/c166/c166demos/src"'];
11
12  make_rtw(user_srcs_arg, user_includes_arg);
13

```

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

- 9 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.

- 10 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).

- 11** 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.

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

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

If you are using a Phytex 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.mdl`. This model is configured to launch the debugger at the end of the build. Included is a comparison with another custom storage class variable in `c166_user_io.mdl`

# Specifying C166 Microcontroller Bit-Addressable Memory

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

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

- 1 Ensure that the signal has the Simulink 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.

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

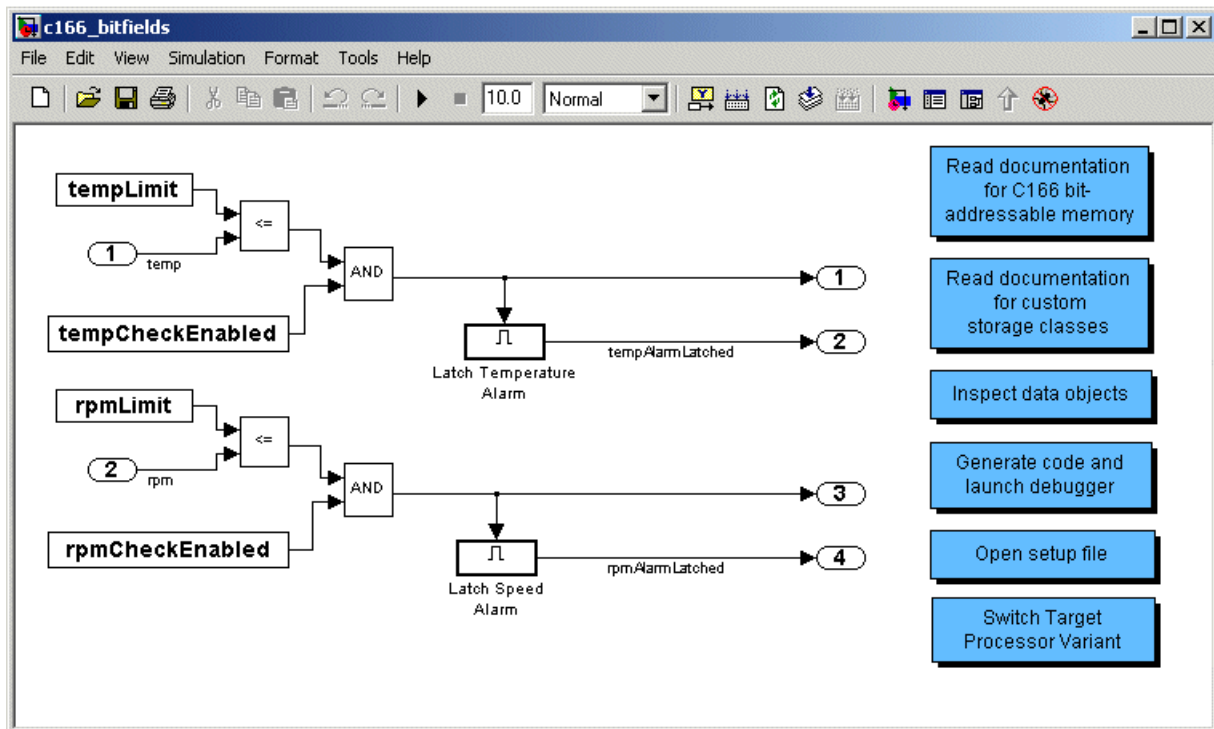
One of the signals in the demo model `c166_user_io.mdl` also uses the custom storage class `SimulinkC166.Signal` to specify that the signal uses bit-addressable memory. You can compare this with the `c166_bitfields` example; it is included in the steps in “Using the Bitfield Example Model” on page 4-3.

## Using the Bitfield Example Model

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

Follow these steps:

- 1 Open `c166_bitfields.mdl`.



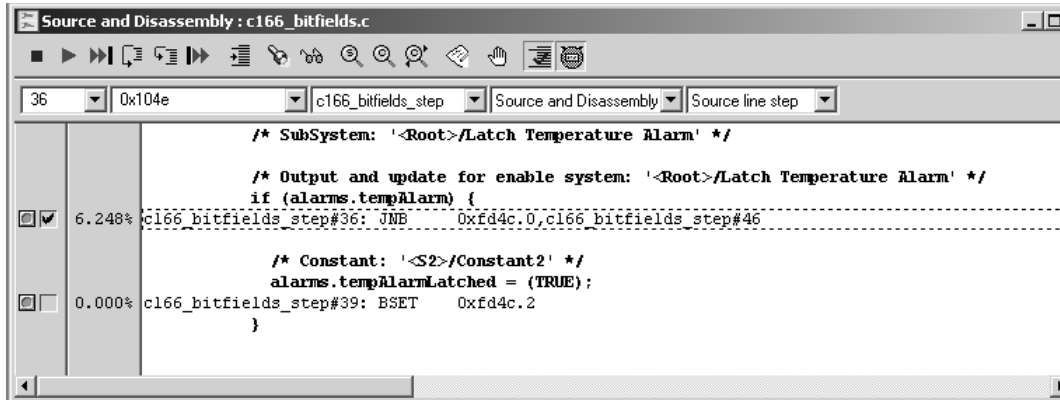
- 2 Double-click **Generate code and launch debugger**.

Code is generated and the debugger is started.

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

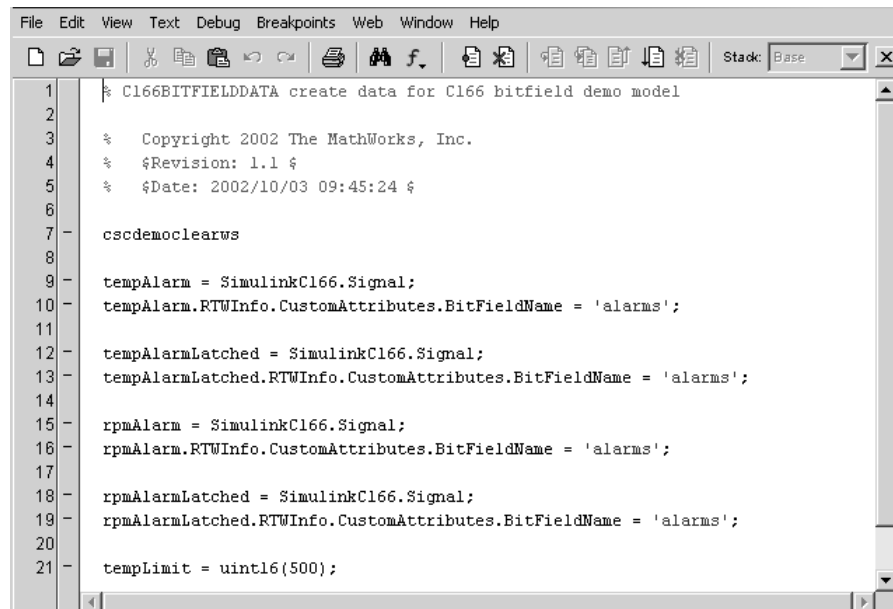
The example following shows a sample of the generated code.

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



```
/* SubSystem: '<Root>/Latch Temperature Alarm' */  
  
/* Output and update for enable system: '<Root>/Latch Temperature Alarm' */  
if (alarms.tempAlarm) {  
6.248% c166_bitfields_step#36: JNB 0xfd4c.0,c166_bitfields_step#46  
-----  
/* Constant: '<S2>/Constant2' */  
alarms.tempAlarmLatched = (TRUE);  
0.000% c166_bitfields_step#39: BSET 0xfd4c.2  
}
```

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

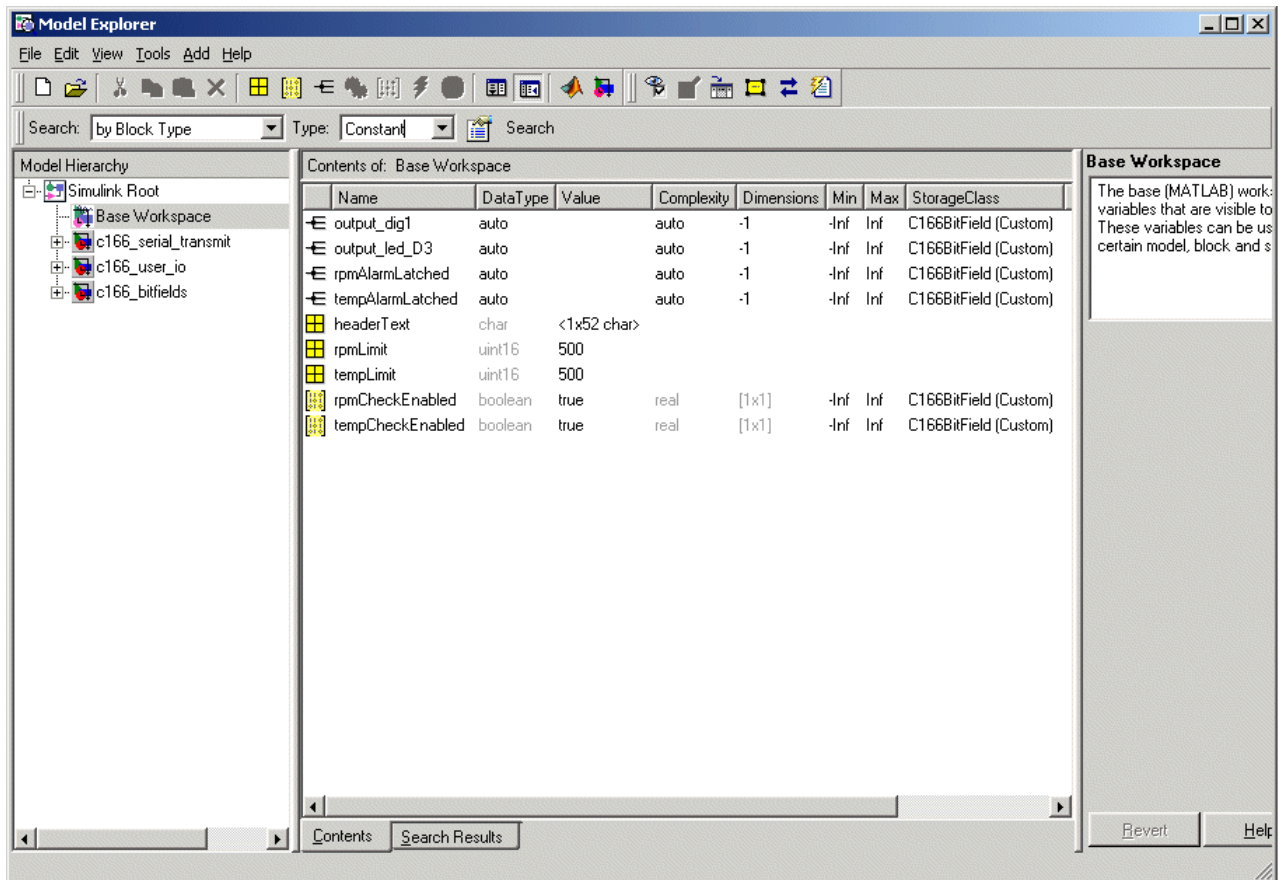


```
File Edit View Text Debug Breakpoints Web Window Help  
C166BITFIELDDATA create data for C166 bitfield demo model  
  
% Copyright 2002 The MathWorks, Inc.  
% $Revision: 1.1 $  
% $Date: 2002/10/03 09:45:24 $  
  
csdcmoclearws  
  
tempAlarm = SimulinkC166.Signal;  
tempAlarm.RTWInfo.CustomAttributes.BitFieldName = 'alarms';  
  
tempAlarmLatched = SimulinkC166.Signal;  
tempAlarmLatched.RTWInfo.CustomAttributes.BitFieldName = 'alarms';  
  
rpmAlarm = SimulinkC166.Signal;  
rpmAlarm.RTWInfo.CustomAttributes.BitFieldName = 'alarms';  
  
rpmAlarmLatched = SimulinkC166.Signal;  
rpmAlarmLatched.RTWInfo.CustomAttributes.BitFieldName = 'alarms';  
  
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 general documentation for custom storage classes** in the model to go directly to the relevant Real-Time Workshop Embedded Coder help section.

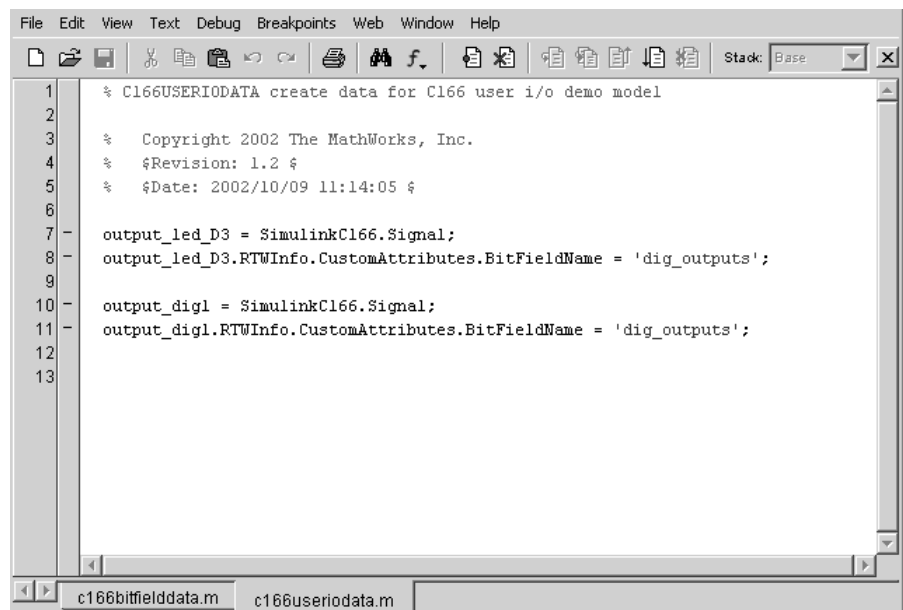
- 5 You can double-click **Inspect data objects** to inspect all the Simulink data objects that are available to the model.



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.mdl`.
- 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
% C166USERIODATA create data for C166 user i/o demo model
% Copyright 2002 The MathWorks, Inc.
% $Revision: 1.2 $
% $Date: 2002/10/09 11:14:05 $
output_led_D3 = SimulinkC166.Signal;
output_led_D3.RTWInfo.CustomAttributes.BitFieldName = 'dig_outputs';
output_dig1 = SimulinkC166.Signal;
output_dig1.RTWInfo.CustomAttributes.BitFieldName = 'dig_outputs';
```

Compare with `c166bitfielddata.m`.

For more details on the variables in this model, see “Tutorial: Using the Example Driver Functions” on page 3-10.



# Execution Profiling

---

This section contains the following topics:

“Overview of Execution Profiling” (p. 5-2)	The steps involved in performing execution-profiling analysis on a model.
“Real-Time Workshop Options for Execution Profiling” (p. 5-4)	How to configure options for execution profiling.
“Multitasking Demo Model” (p. 5-7)	Step-by-step-instructions for running the multitasking demo and interpreting the execution profiling results.

## 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 ASCO 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 5-4.
- 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`.

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.

## 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.

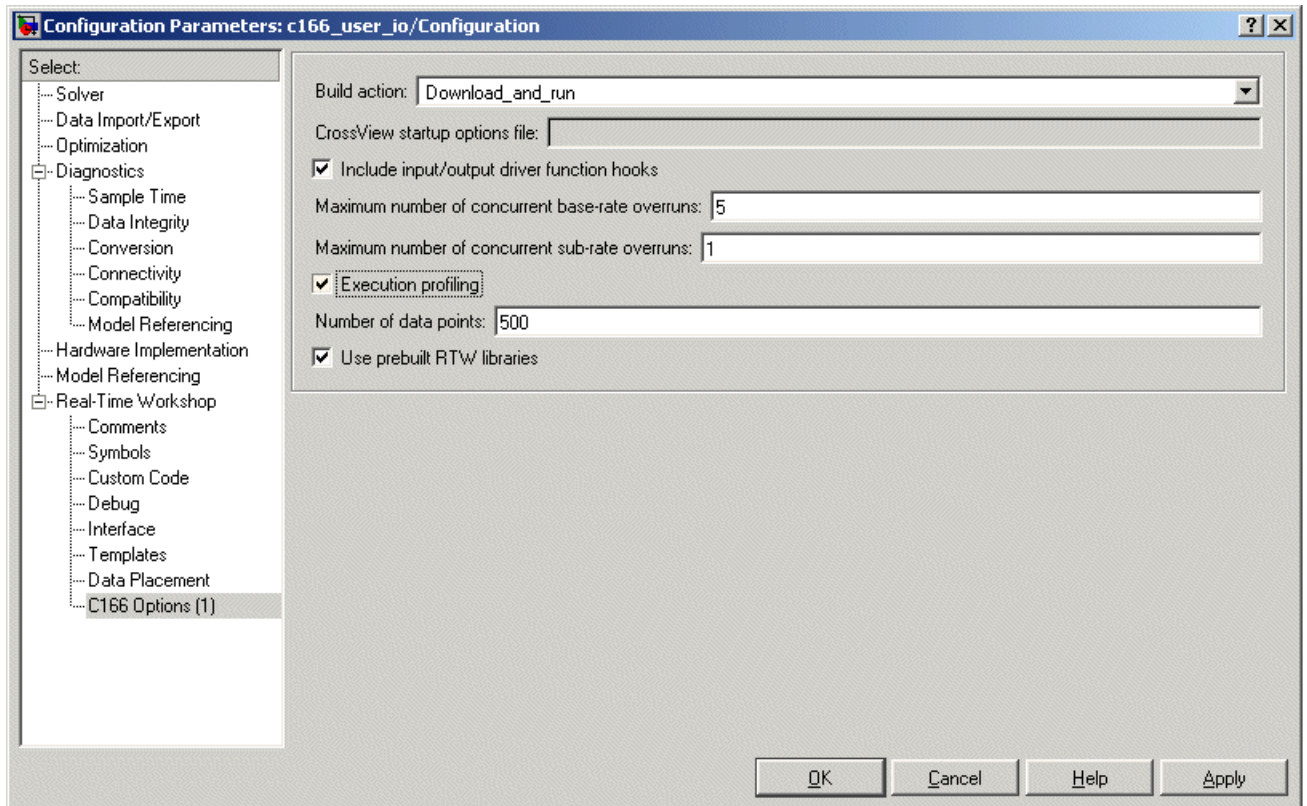
## Execution Profiling Blocks

See the block reference sections:

- C166 Execution Profiling via ASCO
- 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.

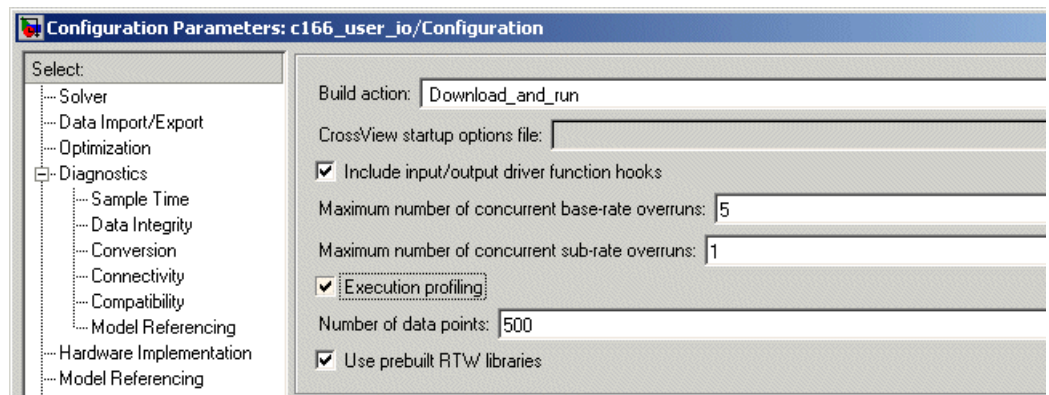
When code for the model is generated, these functions will update data on the worst-case turnaround time for each timer-based task as well as the worst-case number of concurrent task overruns, whenever a previous worst-case value is exceeded. Additionally, when a trigger is provided, data will be logged 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 ASCO, 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.

## Multitasking Demo Model

The demo model `c166_multitasking.mdl` 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 5-5).

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.mdl

If viewing in the Help browser, you can click the link to open the model.

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

The Configuration Parameters dialog appears.

**3** Select **C166 Options (1)** under **Real-Time Workshop** in the tree.

**4** Ensure the **Build action** is `Download_and_run` , and click **OK** to dismiss the dialog.

**5** 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.

**6** 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, then the CrossView Pro Debugger starts, connects to the target, and downloads the code.

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

**8** 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.

**9** 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



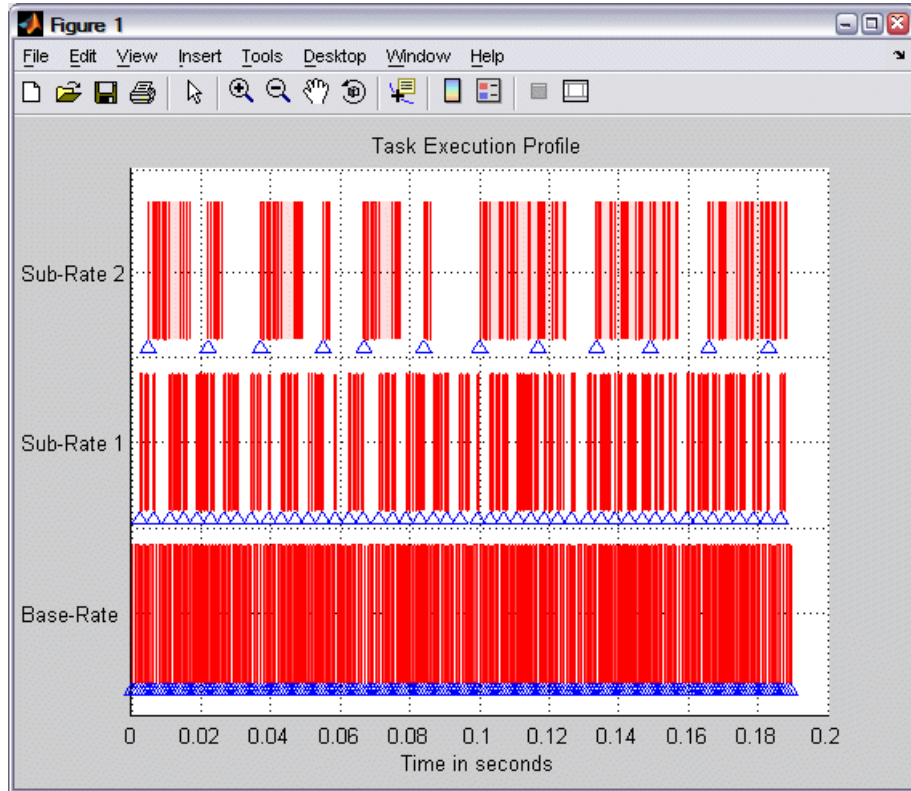
switch to CAN, and follow the same instructions as for a serial connection, except step 7 when the application is running. At the command line, type

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

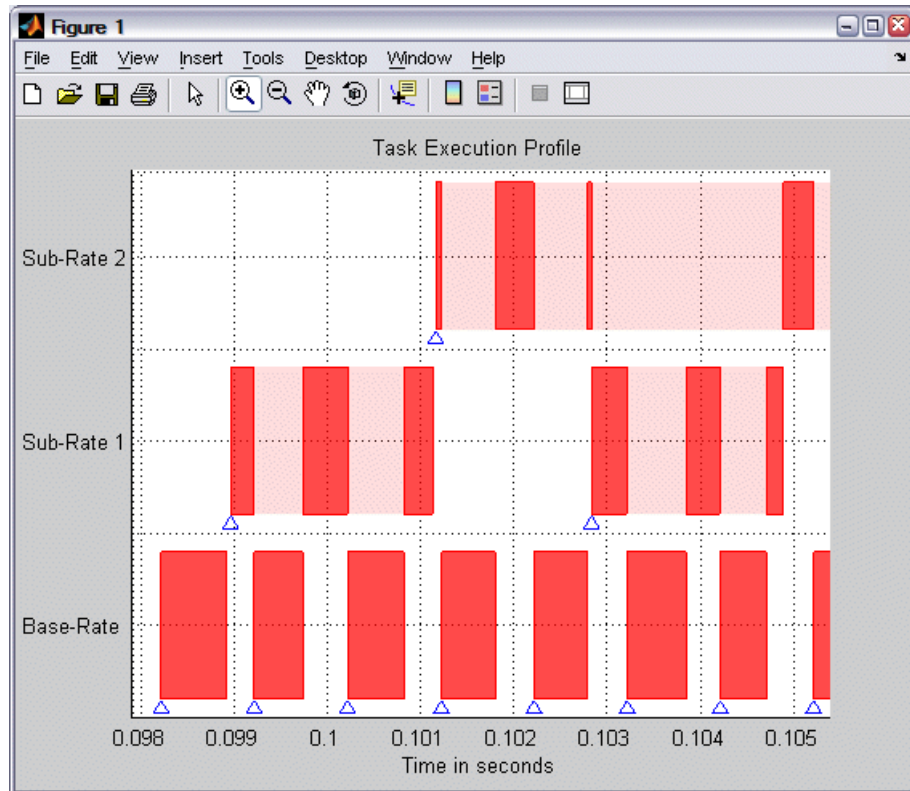
You will see command line messages as the function tests the CAN channel, and requests and collects profiling data. When using CAN, it can be useful to run a monitor program such as `btest32` to verify that the model is running — for example you will see messages appearing on the CAN bus and you can 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 5-3 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

# Blocks — Categorical List

---

“Embedded Target for Infineon C166 Microcontrollers Block Library” (p. 6-2)	An overview of the C166 block library structure.
“C166 Drivers Library” (p. 6-3)	Category tables for the blocks contained in the C166 Drivers library and sublibraries.

## Embedded Target for Infineon C166 Microcontrollers Block Library

The Embedded Target for Infineon C166 Microcontrollers provides one block library, containing seven sublibraries that support different functions. The tables below reflect that organization:

- “C166 Drivers Library” on page 6-3
  - Asynchronous/Synchronous Serial Interface Sublibrary on page 6-3
  - CAN Interface Sublibrary on page 6-4
  - Execution Profiling Sublibrary on page 6-4
  - TwinCAN Interface Sublibrary on page 6-5
  - Interrupts Sublibrary on page 6-6
  - Utilities Sublibrary on page 6-6
  - Digital Input/Output Sublibrary on page 6-6

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

## C166 Drivers Library

### Top-Level Library

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

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/hardware simulator. The C166 Resource Configuration block is required if there are device driver blocks in the model.

---

**Note** 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.

---

### Asynchronous/Synchronous Serial Interface Sublibrary

Block Name	Purpose
Serial Receive	Configure C166 microcontroller for serial receive
Serial Transmit	Configure C166 microcontroller for serial transmit

### CAN Interface Sublibrary

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

You can also use the CAN message blocks that are part of the CAN Blockset. See the CAN Blockset Reference for the following blocks:

Block Name	Purpose
CAN Message Filter	Dispatch message processing based on message ID
CAN Message Packing	Map Simulink signals to CAN messages
CAN Message Unpacking	Inspect and unpack the individual fields in a CAN message

### Execution Profiling Sublibrary

Block Name	Purpose
C166 Execution Profiling via ASCO	Provide a serial interface to the execution profiling engine



**Execution Profiling Sublibrary (Continued)**

<b>Block Name</b>	<b>Purpose</b>
C166 Execution Profiling via CAN A	Provide a CAN interface to the execution profiling engine via CAN channel A
C166 Execution Profiling via TwinCAN A	Provide a CAN interface to the execution profiling engine via TwinCAN channel A, for XC16x variants of the Infineon C166 microprocessor

**TwinCAN Interface Sublibrary**

<b>Block Name</b>	<b>Purpose</b>
CAN Calibration Protocol (C166, TwinCAN)	Implement the CAN Calibration Protocol (CCP) standard for XC16x variants of the Infineon C166 microprocessor
TwinCAN Bus Status	Output the Bus Off or Error Warning state of a CAN node on XC16x variants of the Infineon C166 microprocessor
TwinCAN Receive	Receive CAN messages via the TwinCAN module on XC16x variants of the Infineon C166 microprocessor
TwinCAN Reset	Reset a CAN node on XC16x variants of the Infineon C166 microprocessor
TwinCAN Transmit	Transmit CAN messages from the TwinCAN module on XC16x variants of the Infineon C166 microprocessor

### Interrupts Sublibrary

Block Name	Purpose
Fast External Interrupt	Generate an asynchronous function-call trigger when an interrupt occurs

### Utilities Sublibrary

Block Name	Purpose
Switch Target Configuration	Configure your model and Target Preferences to one of a set of predefined hardware configurations

### Digital Input/Output Sublibrary

Block Name	Purpose
Digital In	Digital input driver that reads the value of a specified port/pin number
Digital Out	Digital output driver that sets the logical state of the specified pin

## Configuration Class Blocks

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



---

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

---

## Using Block Reference Pages

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

- **Purpose** — describes why you use the block or function.
- **Library** — identifies the block library where you find the block.
- **Description** — describes what the block does.
- **Dialog Box** — shows the Block Parameters dialog and describes the parameters and options contained in the dialog. Each parameter or option appears with the appropriate choices and effects.



# Blocks — Alphabetical List

---

# C166 Execution Profiling via ASCO

---

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

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

**Description** The C166 Execution Profiling via ASCO 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 (ASCO). See also the MATLAB command `profile_c166`.



`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.

```
profdata = profile_c166(connection)
```

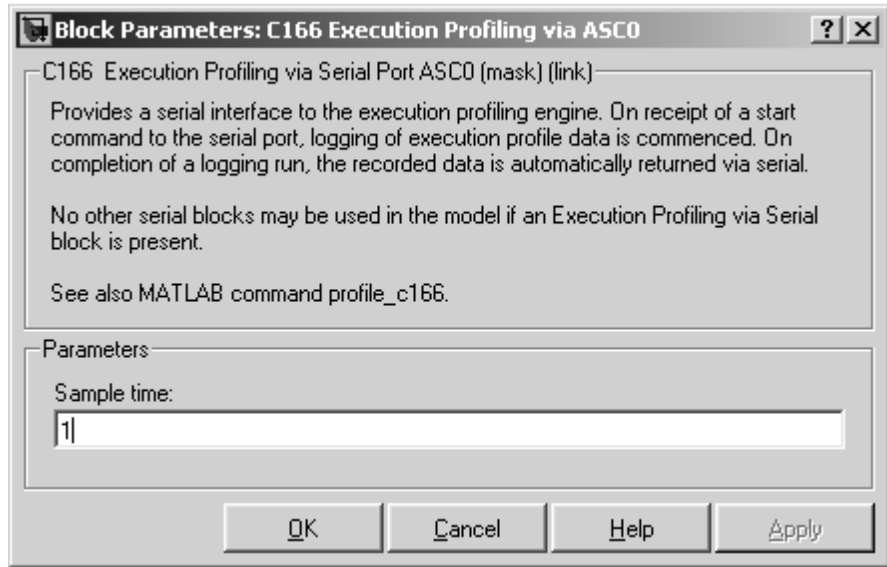
returns the execution profiling data in the format documented by `exprofile_unpack`.

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.mdl`.

## Dialog Box



### 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.

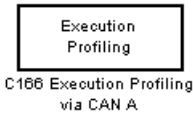
# C166 Execution Profiling via CAN A

---

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

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

**Description** 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.

```
profdata = profile_c166(connection)
```

returns the execution profiling data in the format documented by `exprofile_unpack`.

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.

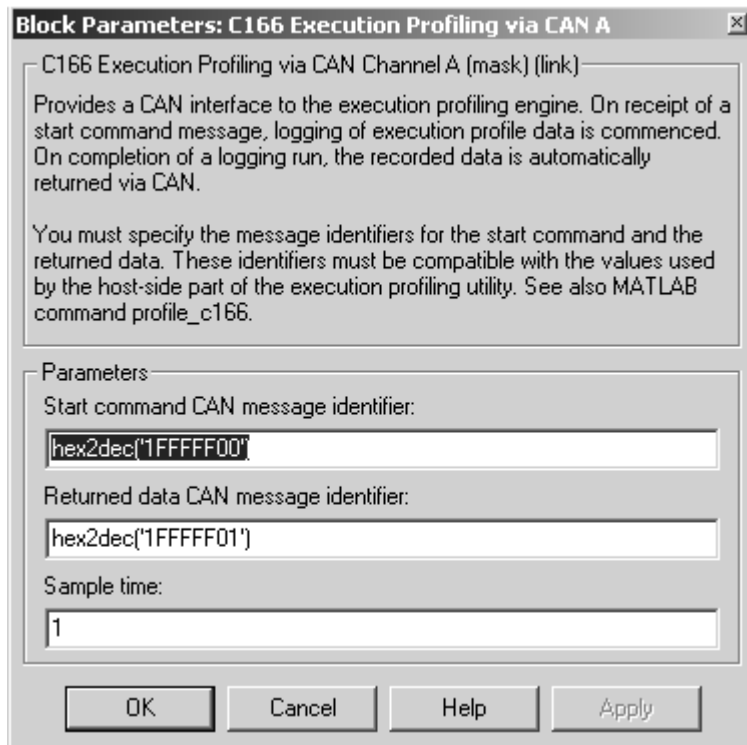


# C166 Execution Profiling via CAN A

- 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.mdl`.

## Dialog Box



### 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

# C166 Execution Profiling via CAN A

---

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.

# C166 Execution Profiling via TwinCAN A

---

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

**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.



C166 Execution Profiling  
via TwinCAN A

# C166 Resource Configuration

---

<b>Purpose</b>	Support device configuration for the C166 microcontroller
<b>Library</b>	Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library
<b>Description</b>	<p>The C166 Resource Configuration block differs in function and behavior from conventional blocks. Therefore, we refer to this block as the C166 Resource Configuration <i>object</i>.</p> <p>The C166 Resource Configuration object is required to provide information that is used to configure driver blocks and timer interrupts.</p> <ul style="list-style-type: none"><li>• You must include this block in your model if<ul style="list-style-type: none"><li>▪ You are using any of the driver blocks supplied with Embedded Target for Infineon C166 Microcontrollers</li><li>▪ You are taking advantage of the automatically generated scheduler that is driven by timer interrupts.</li></ul></li><li>• 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 <code>main.c</code>).</li></ul> <p>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.</p> <p>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.</p>

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.

---

## Types of Configurations

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

- C166 Drivers Configuration: C166 microcontroller clocks and other CPU-related parameters
- Asynchronous/Synchronous Serial Interface Configuration: parameters related to the serial driver blocks and Simulink external mode
- 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 (ASCO) sublibrary, you

# C166 Resource Configuration

---

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.

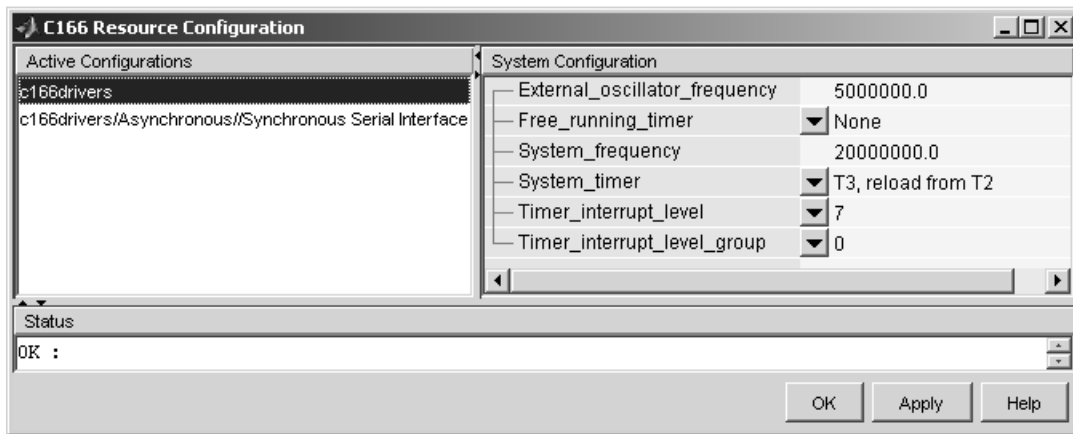
A configuration remains active until all blocks associated with it are removed from the model or subsystem. At that point, the configuration is in an *inactive* state. Inactive configurations are lost from the C166 Resource Configuration window when the model is saved and reopened. You can reactivate 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`.

# C166 Resource Configuration



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.

# C166 Resource Configuration

---

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

---

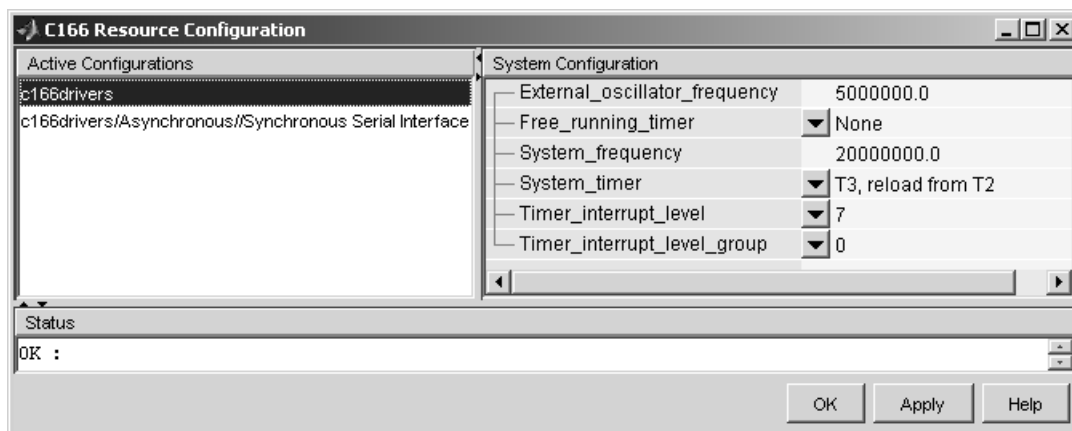
- **Status** panel: The **Status** panel displays error messages that may arise if resource allocation conflicts are detected in the configuration.
- **OK** button: Dismisses the window.

## 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**

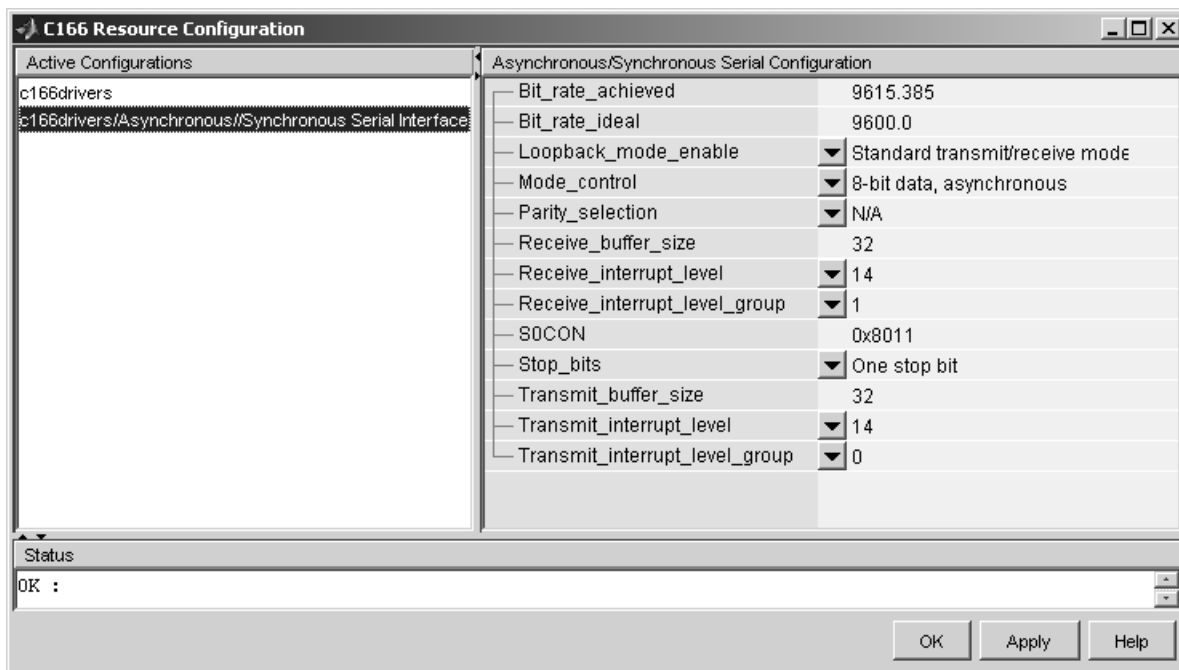
You must select which timer to use for generating interrupts to drive the model update rate. You should select a timer, or timer pair, that you do not intend to use for any other purpose within your application. We recommend you choose a pair of timers, e.g., T6, with reload from CAPREL. This will give 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.

# C166 Resource Configuration

## 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 SOBG and bitfield SOBRS of register SOCON.

### **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.

**SOCON**

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

**Stop\_bits**

You must select either 1 or 2 stop bits.

**Transmit\_buffer\_size**

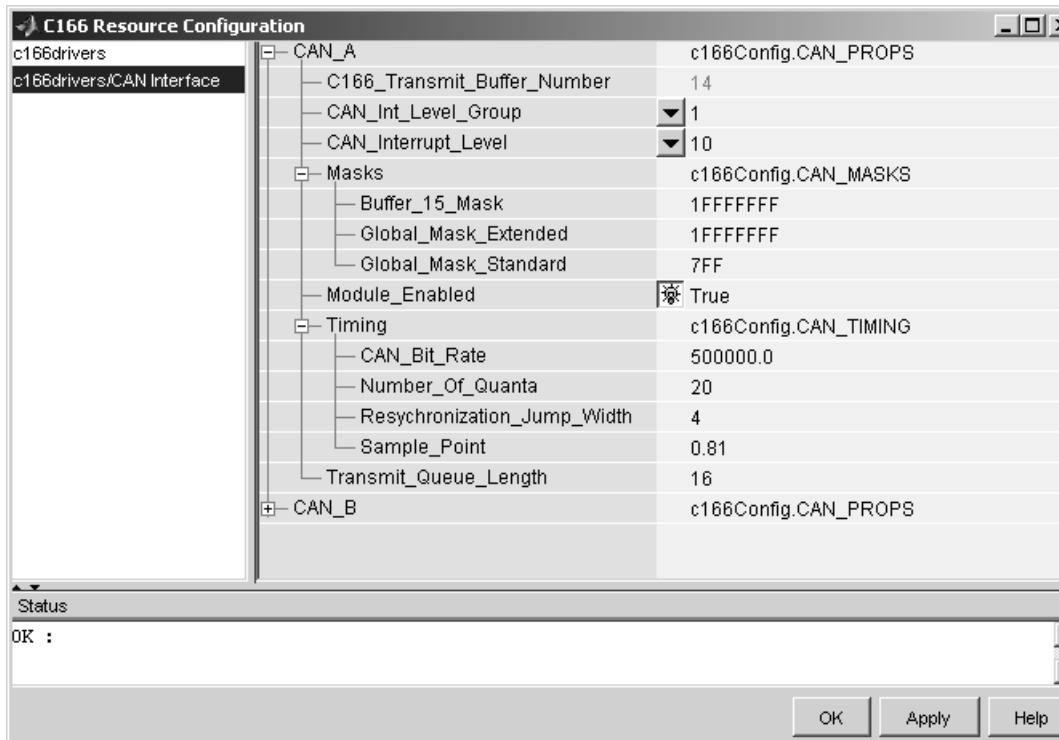
See Receive\_buffer\_size.

**Transmit\_interrupt\_level and Transmit\_interrupt\_level\_group**

See Receive parameters above.

# C166 Resource Configuration

## 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\_Interruption\_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 a 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.

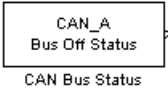
### **Transmit\_Queue\_Length**

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.

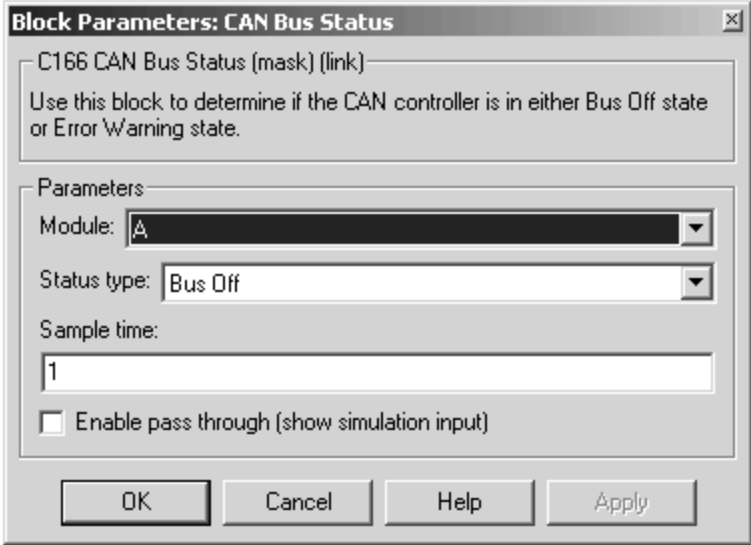
**Purpose** Output the Bus Off or Error Warning state of a CAN module

**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.



## 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.



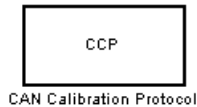
## Purpose

Implement the CAN Calibration Protocol (CCP) standard

## Library

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

## Description



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 2-16).

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.

# CAN Calibration Protocol (C166)

---

Using DAQ lists for signal monitoring has the following advantages over the polling method:

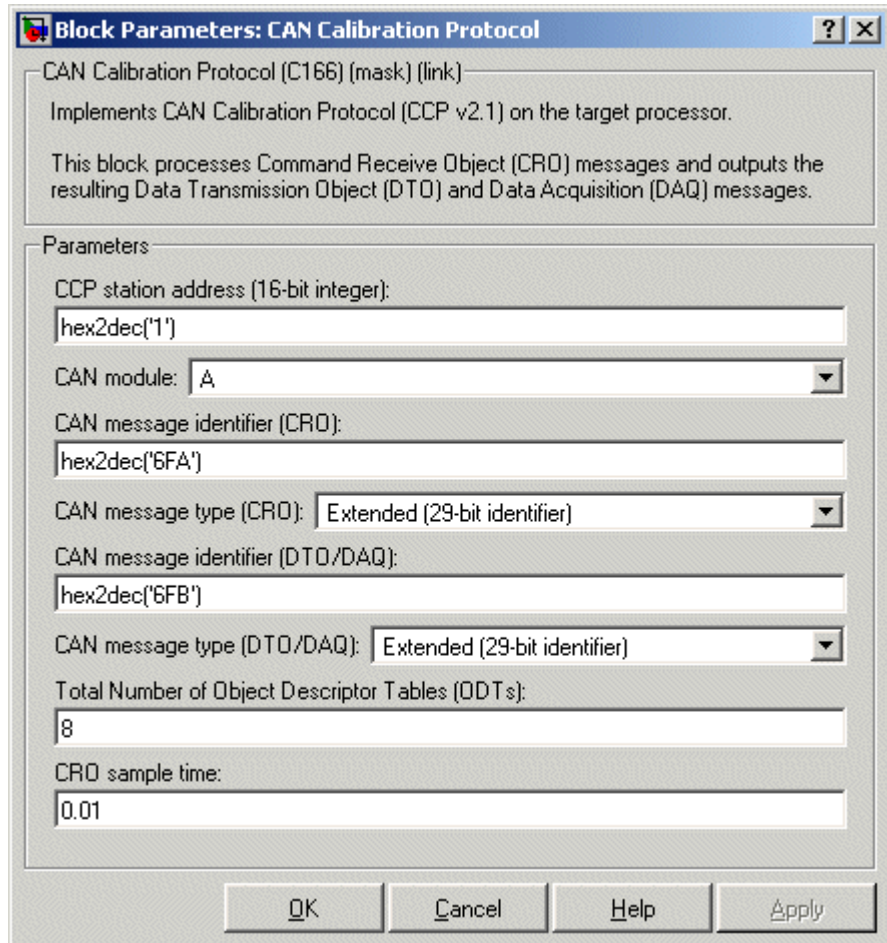
- There is no need for the host to poll for the data. Network traffic is halved.
- The data is transmitted at the correct update rate for the signal. Therefore, there is no unnecessary network traffic generated.
- Data is guaranteed to be consistent. The transmission takes place after the signals have 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** [?] [X]

CAN Calibration Protocol (C166) (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

# CAN Calibration Protocol (C166)

---

- 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).

# CAN Calibration Protocol (C166, TwinCAN)

---

## Purpose

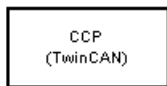
Implement the CAN Calibration Protocol (CCP) standard for XC16x variants of the Infineon C166 microprocessor

## 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.



CAN Calibration Protocol

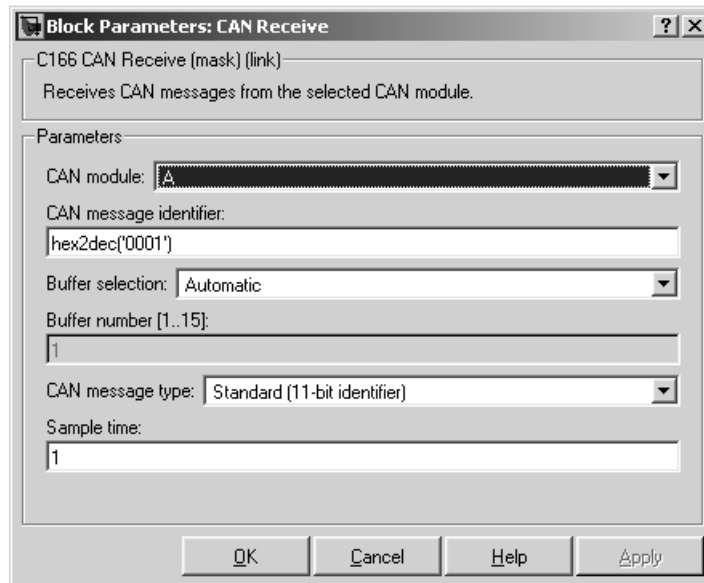
# CAN Receive

**Purpose** Receive CAN messages from the CAN module on the 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

---

---

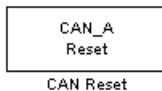
**Note** The CAN Receive block sample time must be set to a value that is smaller than the minimum time between CAN messages that will be received into the corresponding buffer. If more than one message is received into a buffer during a single sample interval, the older message will be overwritten.

---

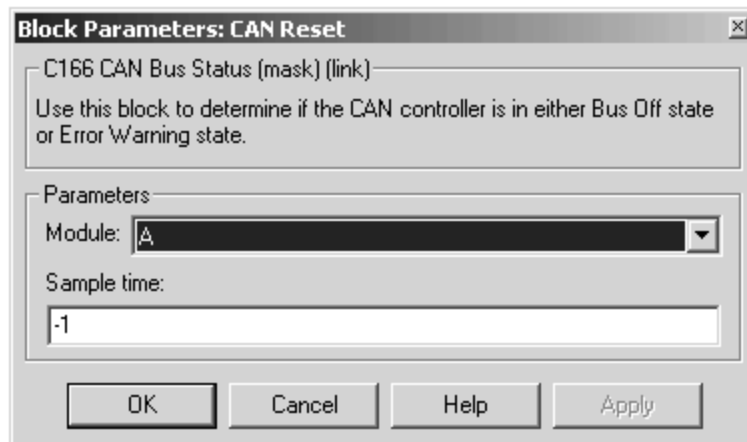
**Purpose** Reset a CAN module

**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).



## Dialog Box



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

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

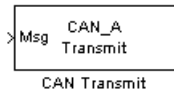
# CAN Transmit

---

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

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

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



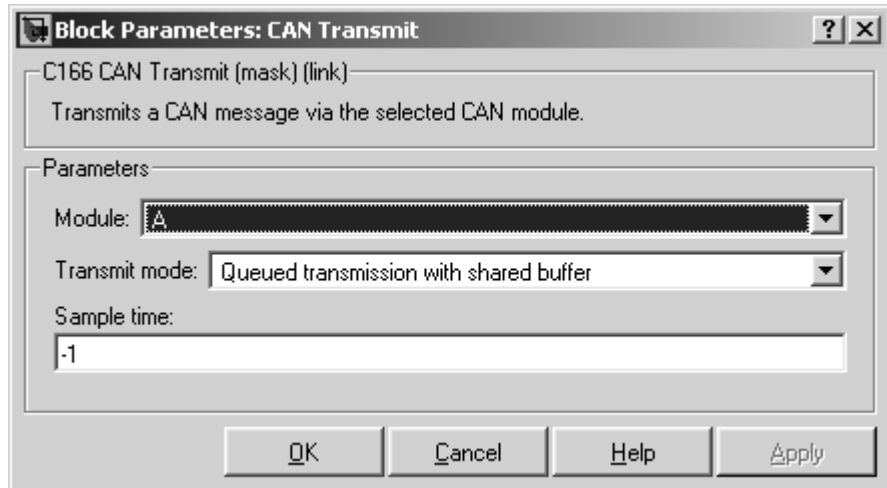
The default mode is to use a priority-based message queue shared by all transmit blocks operating in this mode; the priority-based message queue operates with CAN buffer 14; when a message is successfully transmitted from this buffer, an interrupt is generated and the highest priority message from the 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,

# CAN Transmit

---

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.

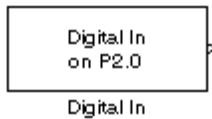
## **Sample time**

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 the value of a specified port/pin number

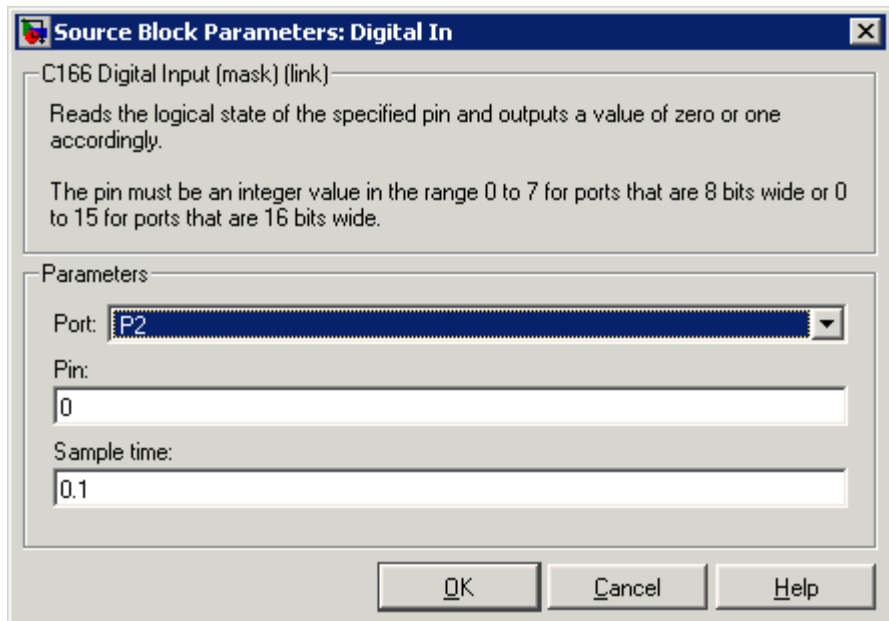
**Library** Embedded Target for Infineon C166 Microcontrollers/C166 Driver Library/Digital Input/Output

## Description



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

## Dialog Box



**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**

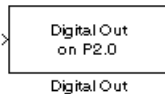
The time interval between samples. The default is 0.1. See “Specifying Sample Time” in the Simulink documentation for more information.



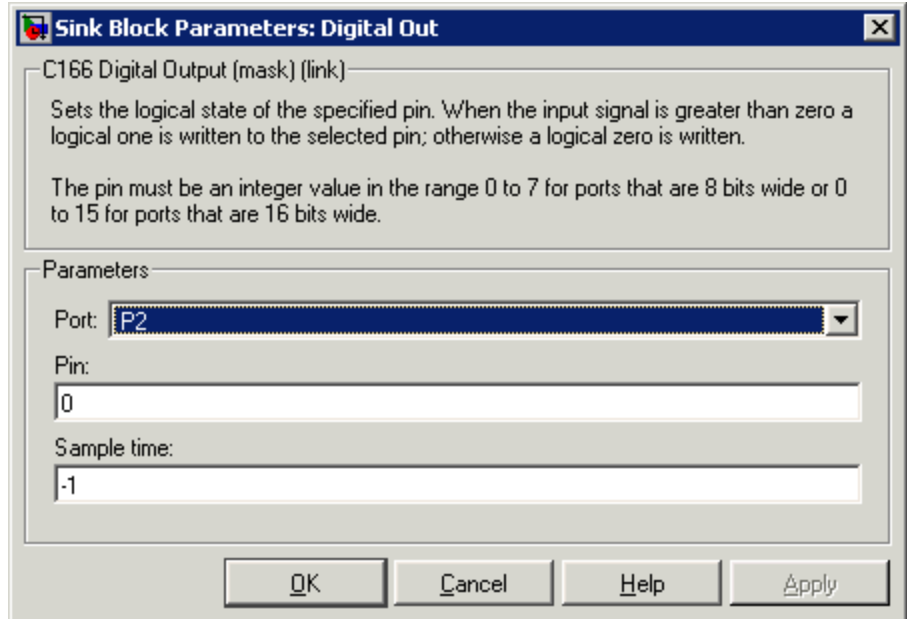
**Purpose** Digital output driver that sets the logical state of the specified pin

**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.



## Dialog Box



**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.

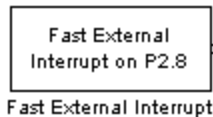
## **Sample time**

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

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

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

## Description



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.

## 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.

# Fast External Interrupt

---

## **Show simulation input**

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

<b>Purpose</b>	Configure C166 microcontroller for serial receive
<b>Library</b>	Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ Asynchronous/Synchronous Serial Interface
<b>Description</b>	<p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p>

# 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.

---

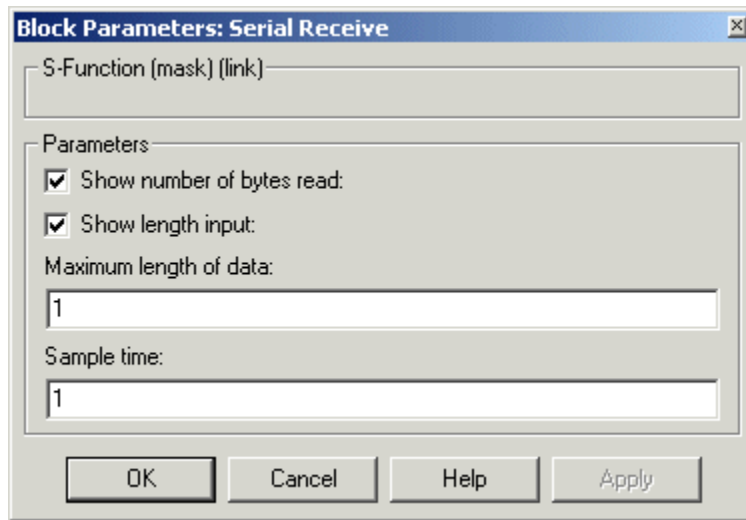
## Block Inputs and Outputs

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

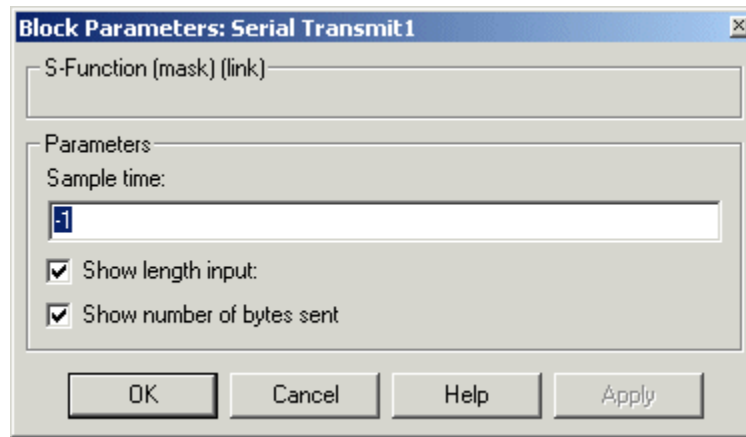
The first input contains the data to be transmitted; this input signal may be either a vector or scalar with data type `uint8`.

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

## Dialog Box



### 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 input; if selected, the second input is enabled, which controls the number of bytes to send.

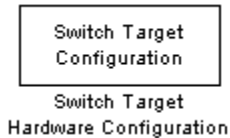
### Show number of bytes sent

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

**Purpose** Configure your 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



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. If your setup does not correspond to one of the predefined configurations, you may wish to use the file (c166switchconfig.m) as a template for setting up your own customized configurations. The predefined configurations include settings for

- Phytex phyCORE-C167CS (RAM) [2]
- Phytex phyCORE-C167CS (flash)
- Infineon XC167CI Starter Kit
- Phytex phyCORE-ST10F269
- Phytex kitCON-C167CR (RAM)
- Phytex kitCON-C167CR (flash)

# TwinCAN Bus Status

---

<b>Purpose</b>	Output the Bus Off or Error Warning state of a CAN node on XC16x variants of the Infineon C166 microprocessor
<b>Library</b>	Embedded Target for Infineon C166 Microcontrollers/ C166 Driver Library/ TwinCAN Interface
<b>Description</b>	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.

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

**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

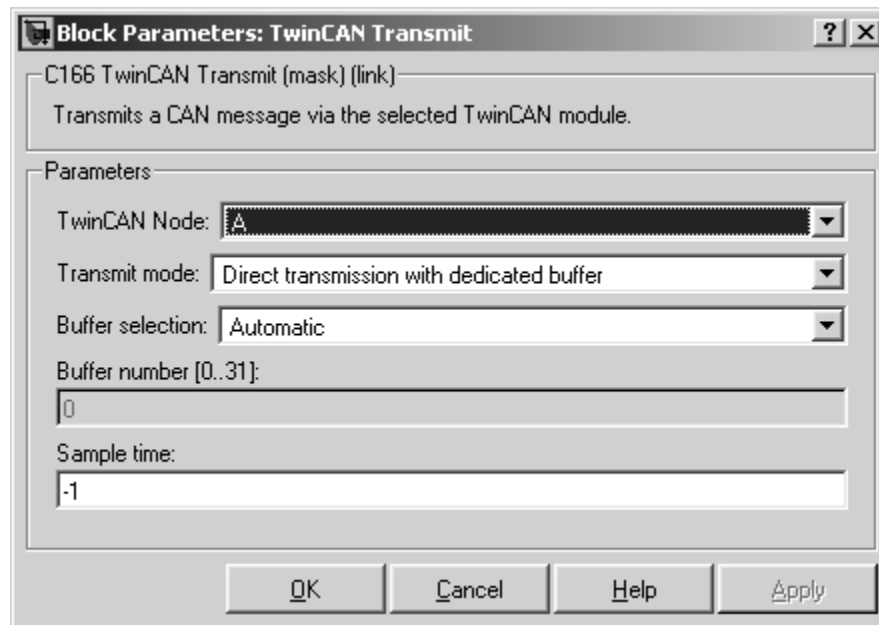
---

<b>Purpose</b>	Reset a CAN node on XC16x variants of the Infineon C166 microprocessor
<b>Library</b>	Embedded Target for Infineon C166 Microcontrollers/C166 Driver Library/TwinCAN Interface
<b>Description</b>	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.

<b>Purpose</b>	Transmit CAN messages from the TwinCAN module on XC16x variants of the Infineon C166 microprocessor
<b>Library</b>	Embedded Target for Infineon C166® Microcontrollers/ C166 Driver Library/ TwinCAN Interface
<b>Description</b>	<p>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.</p> <p>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.</p> <p>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.</p> <p>The TwinCAN Transmit block should be connected to CAN Message Packing/Unpacking blocks. Do not ground the block or leave it unconnected.</p>

# TwinCAN Transmit

## Dialog Box



### **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.



## A

ASAP2 files  
    generating for C166 2-14  
ASAP2 files, generating 2-16

## B

bit-addressable memory 4-1  
blocks  
    C166 Execution Profiling via ASCO 7-2  
    C166 Execution Profiling via CAN A 7-4  
    C166 Execution Profiling via TwinCAN  
        A 7-7  
    C166 Resource Configuration 7-8  
    CAN Bus Status 7-19  
    CAN Calibration Protocol (C166) 7-21  
    CAN Calibration Protocol (C166,  
        TwinCAN) 7-27  
    CAN Receive 7-28  
    CAN Reset 7-31  
    CAN Transmit 7-32  
    Digital In 7-35  
    Digital Out 7-37  
    Fast External Interrupt 7-39  
    Serial Receive 7-41  
    Serial Transmit 7-44  
    Switch Target Configuration 7-47  
    TwinCAN Bus Status 7-48  
    TwinCAN Receive 7-49  
    TwinCAN Reset 7-50  
    TwinCAN Transmit 7-51

## C

C166 Execution Profiling via ASCO block 7-2  
C166 Execution Profiling via CAN A block 7-4  
C166 Execution Profiling via TwinCAN A  
    block 7-7  
C166 Resource Configuration block 7-8  
C166 Target 1-1

CAN Bus Status block 7-19  
CAN Calibration Protocol (C166) block 7-21  
CAN Calibration Protocol (C166, TwinCAN)  
    block 7-27  
CAN Receive block 7-28  
CAN Reset block 7-31  
CAN Transmit block 7-32  
Configuration Class blocks 6-6  
custom storage class 4-1

## D

device driver blocks  
    C166 Digital In 7-35  
    C166 Digital Out 7-37  
    C166 Execution Profiling via ASCO 7-2  
    C166 Execution Profiling via CAN A 7-4  
    C166 Execution Profiling via TwinCAN  
        A 7-7  
    C166 Resource Configuration 7-8  
    C166 Serial Receive 7-41  
    C166 Serial Transmit 7-44  
    CAN Bus Status 7-19  
    CAN Calibration Protocol (C166) 7-21  
    CAN Calibration Protocol (C166,  
        TwinCAN) 7-27  
    CAN Receive 7-28  
    CAN Reset 7-31  
    CAN Transmit 7-32  
    Digital In 7-35  
    Digital Out 7-37  
    Fast External Interrupt 7-39  
    Serial Receive 7-41  
    Serial Transmit 7-44  
    Switch Target Configuration 7-47  
    TwinCAN Bus Status 7-48  
    TwinCAN Receive 7-49  
    TwinCAN Reset 7-50  
    TwinCAN Transmit 7-51  
Digital In block 7-35

Digital Out block 7-37  
downloading code 2-6

## **E**

Embedded Target for Infineon C166  
Microcontrollers  
feature summary 1-3  
example model  
c166\_bitfields 4-1  
c166\_fuelsys 2-14  
c166\_multitasking 5-1  
c166\_serial\_io 2-9  
c166\_serial\_transmit 2-4  
c166\_user\_io 3-1  
execution profiling 5-1

## **F**

Fast External Interrupt block 7-39  
fixed-point example 2-14

## **G**

generating code 2-6

## **I**

installation of Embedded Target for Infineon  
C166 Microcontrollers 1-8  
integrating hand-coded device drivers 3-1

## **M**

multitasking 5-1

## **R**

real-time target  
C166 tutorial 2-2

## **S**

Serial Receive block 7-41  
Serial Transmit block 7-44  
Switch Target Configuration block 7-47

## **T**

TwinCAN Bus Status block 7-48  
TwinCAN Receive block 7-49  
TwinCAN Reset block 7-50  
TwinCAN Transmit block 7-51