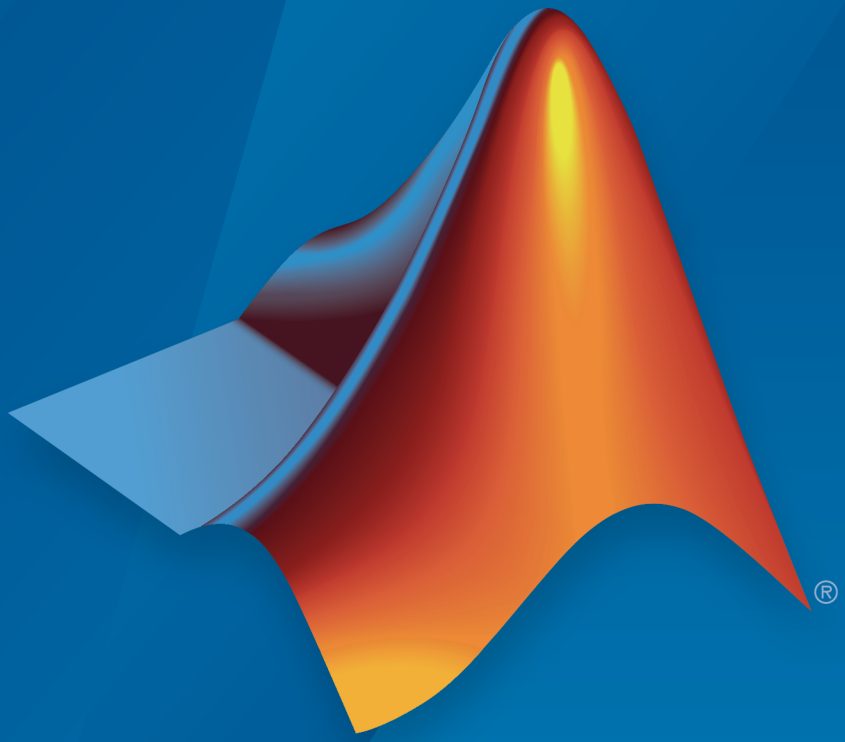
Simulink® Real-Time™

Device Drivers



MATLAB® & SIMULINK®

R2015b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

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

Simulink® Real-Time™ Device Drivers Guide

© COPYRIGHT 2007–2015 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.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

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

Revision History

March 2007	Online only	New for Version 3.2 (Release 2007a)
September 2007	Online only	Updated for Version 3.3 (Release 2007b)
March 2008	Online only	Updated for Version 3.4 (Release 2008a)
October 2008	Online only	Updated for Version 4.0 (Release 2008b)
March 2009	Online only	Updated for Version 4.1 (Release 2009a)
September 2009	Online only	Updated for Version 4.2 (Release 2009b)
March 2010	Online only	Updated for Version 4.3 (Release 2010a)
April 2011	Online only	Updated for Version 5.0 (Release 2011a)
September 2011	Online only	Updated for Version 5.1 (Release 2011b)
March 2012	Online only	Revised for Version 5.2 (Release 2012a)
September 2012	Online only	Revised for Version 5.3 (Release 2012b)
March 2013	Online only	Revised for Version 5.4 (Release 2013a)
September 2013	Online only	Revised for Version 5.5 (Release 2013b)
March 2014	Online only	Revised for Version 6.0 (Release 2014a)
October 2014	Online only	Revised for Version 6.1 (Release 2014b)
March 2015	Online only	Revised for Version 6.2 (Release 2015a)
September 2015	Online only	Revised for Version 6.3 (Release 2015b)

Contents

1 Customizing Simulink Real-Time Drivers

Custom Simulink Real-Time Drivers 1-2
- Simulink Real-Time Drivers 1-2
- When to Write Your Own Drivers 1-2
- Restrictions on Customizing Simulink Real-Time Drivers . . . 1-3
- Expected Background 1-3
- Resources for Customizing Simulink Real-Time Drivers 1-4
- What Makes Up a Simulink Real-Time Driver? 1-6

Before You Start 1-8
- Introduction 1-8
- Driver Types 1-9
- Bus Types and Register Access 1-9
- Register Access 1-10
- Inlining Simulink Real-Time Drivers 1-10

Creating a Custom Driver 1-11

Debugging Notes 1-15

2 PCI Drivers

PCI Bus Considerations 2-2
- Introduction 2-2
- PCI Configuration Space API 2-3
- Memory-Mapped Accesses 2-5
- I/O Port Accesses 2-6

Sample PCI Device Driver 2-7

3 ISA and PC/104 Drivers

ISA and PC/104 Bus Considerations **3-2**
Introduction **3-2**
I/O Mapped **3-2**
Memory Mapped **3-2**

4 Masking Drivers

Creating Driver Subsystem Masks **4-2**

Driver Mask Guidelines **4-3**

Cross-Block Checking **4-5**

When You Are Done **4-6**

Sample Driver Mask **4-7**

Configure Parameters & Dialog Pane **4-8**

Configure Initialization Pane **4-10**

Configure Documentation Pane **4-11**

Create Initialization Callback **4-12**

5 Interrupt Support

Use Interrupts in Simulink Real-Time **5-2**

Add Interrupt Support **5-5**
Introduction **5-5**

Guidelines for Creating Interrupt Functions 5-7
Filling in the Driver `board` Structure 5-8

Hook Function Prototypes — Alphabetical List 5-12

6 Custom Simulink Real-Time Driver Notes

S-Function Guidelines 6-2

mdlStart and mdlTerminate Considerations 6-4

DMA Considerations 6-5

Passing Parameters 6-6

Accessing Registers 6-7
I/O Space 6-7
Memory-Mapped Space 6-7

7 Using the Simulink Real-Time Driver Authoring Tool

Driver Authoring Tool Basics 7-2

Generating Custom Driver Templates 7-3
Using the Driver Authoring Tool 7-3
Setting Up Driver Variables 7-3
Saving the Configuration 7-6
Reloading the Configuration 7-6
Creating the C File Template 7-6
Creating a C MEX File for the Driver 7-7
Customizing the Device Driver Mask 7-7

I/O Structures — Alphabetical List
8

I/O Functions — Alphabetical List
9

1

Customizing Simulink Real-Time Drivers

- "Custom Simulink Real-Time Drivers" on page 1-2
- "Before You Start" on page 1-8
- "Creating a Custom Driver" on page 1-11
- "Debugging Notes" on page 1-15

Custom Simulink Real-Time Drivers

In this section...
"Simulink Real-Time Drivers" on page 1-2
"When to Write Your Own Drivers" on page 1-2
"Restrictions on Customizing Simulink Real-Time Drivers" on page 1-3
"Expected Background" on page 1-3
"Resources for Customizing Simulink Real-Time Drivers" on page 1-4
"What Makes Up a Simulink Real-Time Driver?" on page 1-6

Simulink Real-Time Drivers

The Simulink Real-Time software provides device drivers for a variety of third-party boards. Simulink Real-Time users access these drivers as Simulink blocks from the Simulink Real-Time library (`slrtlib`). If you have a board for which the Simulink Real-Time software does not supply a driver, you can write your own. This topic provides guidelines for writing custom Simulink Real-Time device drivers.

The Simulink Real-Time driver library contains drivers that support third-party boards with many I/O capabilities. This includes drivers for different types of I/O boards, including
Analog-to-digital
Digital-to-analog
Audio
Counters
Shared memory

There are also drivers that support particular protocols, including
RS-232, RS-422, RS-485
GPIB
CAN
UDP
ARINC 429
MIL-1553

When to Write Your Own Drivers

Consider writing your own device drivers for the Simulink Real-Time block library if:

- No Simulink Real-Time driver exists for your I/O needs.
- You are unable to use a board that the Simulink Real-Time software supports.
- You need to extend the functionality of an existing Simulink Real-Time driver.
- The MathWorks® Simulink Real-Time team will not write a device driver for your board.

Restrictions on Customizing Simulink Real-Time Drivers

The Simulink Real-Time software has its own kernel, and you will be writing device drivers aimed at that kernel. A Simulink Real-Time driver is therefore different from a driver for another environment, such as Microsoft® Windows®. The Simulink Real-Time kernel is optimized and small, and does not have the operating system layers that traditional kernels do.

The Simulink Real-Time software installs its own kernel on the target computer. This kernel runs to the exclusion of any other operating system. When writing your own driver:

- You cannot use a driver DLL that accompanies the I/O board from the manufacturer. A manufacturer-supplied DLL will have external dependencies that the Simulink Real-Time kernel cannot resolve. The Simulink Real-Time executable will not be able to load the DLL.
- Do not create your own driver DLL.
- If you do not have access to the register programming information, neither you nor MathWorks can write a device driver for the board. If you have access to the source code of an existing driver for the board, you might be able to port it to the Simulink Real-Time kernel.

Expected Background

This topic assumes that you are already knowledgeable about writing device drivers. It describes the steps specific to writing device drivers for the Simulink Real-Time environment. To write your own device drivers for the Simulink Real-Time system, you need the following background:

- Good C programming skills

- Knowledge of how Simulink simulation works, for example, the type and order of calls
- Knowledge of writing S-functions and compiling those functions as C-MEX functions. This includes a comprehensive knowledge of Simulink callback methods and the Simulink `SimStruct` functions.
- Basic knowledge of Simulink Coder™
- Understanding of I/O programming. Because of the real-time nature of the Simulink Real-Time software, you must develop drivers with minimal latency. Most drivers access the I/O module at the lowest possible level (register programming). Therefore, you must understand how to control an I/O module with register information.
- Knowledge of port and memory I/O access over various buses. You need this information to access I/O modules at the register level.
- Knowledge of computer system fundamentals and internals

Resources for Customizing Simulink Real-Time Drivers

This section lists the resources that are available to you from MathWorks.

References

The following MathWorks documentation provides information that you can refer to when customizing Simulink Real-Time drivers:

See...	For...
"Getting Started with Simulink"	Overall description of the Simulink environment and how the Simulink software performs simulations.
"Block Creation"	Description of how to create custom Simulink blocks.
"Introducing MEX Files"	How to write MATLAB® MEX-files.
"Getting Started with Simulink Coder"	Overall description of Simulink Coder fundamentals, and guidelines on understanding I/O boards and low-level programming for drivers for those boards.

MathWorks Consulting

You can alternatively contact the MathWorks Consulting Services Group about the fee-based creation of a driver for your board.

Source Code

You can examine the source code for existing Simulink Real-Time device drivers as a reference for your custom drivers. Refer to the following folder:

matlabroot\toolbox\rtw\targets\xpc\target\build\xpcblocks

In this folder, you might notice that some drivers use outdated Simulink Real-Time driver functions. For the current functions to use, see "Simulink Real-Time Exported Functions" on page 1-5.

Simulink Real-Time Exported Functions

The Simulink Real-Time software provides kernel functions that you can use when writing your device drivers. These functions enable you to input and output data, configure PCI devices, and specify timeout intervals. Use only the functions documented in this topic. The guidelines in this document are not applicable if you are using a Simulink Real-Time software version prior to Simulink Real-Time software version 3.2 (R2007a).

The exported functions are:

- `xpcAllocPhysicalMemory`
- `xpcBusyWait`
- `xpcFreePhysicalMemory`
- `xpcGetElapsedTime`
- `xpcGetPCIDeviceInfo`
- `xpcInpB`
- `xpcInpW`
- `xpcInpDW`
- `xpcIsModelInit`
- `xpcOutpB`
- `xpcOutpW`
- `xpcOutpDW`
- `xpcReserveMemoryRegion`
- `xpcShowPCIDeviceInfo`

- `xpcSubtractTime`

Third-Party Folder

The Simulink Real-Time software provides the following folder to help you integrate your custom driver.

matlabroot`\toolbox\rtw\targets\xpc\target\build\xpcblocks\thirdpartydrivers`

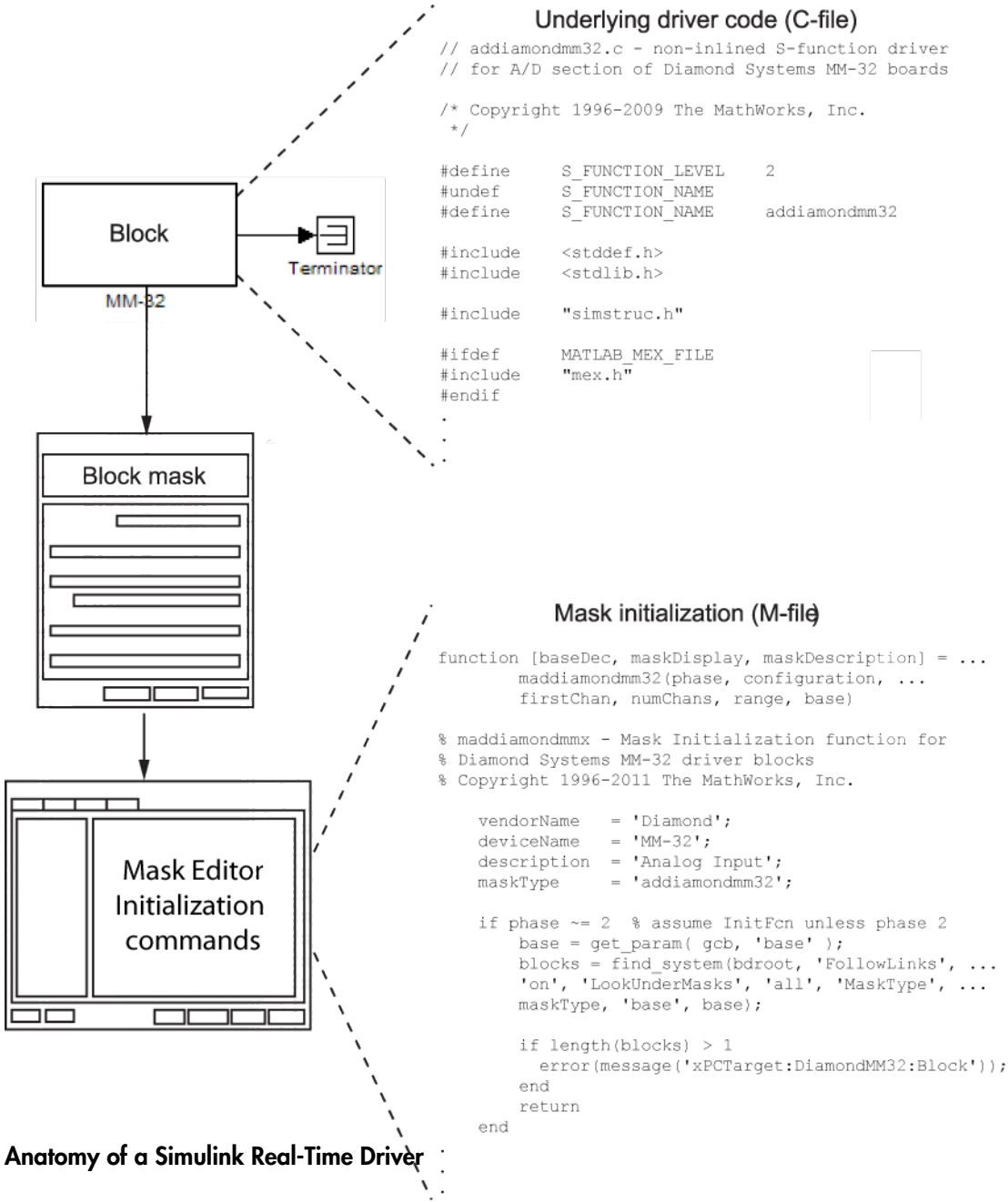
This folder provides template files that you copy and customize for your drivers. Place all files that support your drivers in this folder.

What Makes Up a Simulink Real-Time Driver?

A Simulink Real-Time device driver is an S-function with functions that access an I/O board.

Like any device driver, a Simulink Real-Time driver interfaces between the user and an I/O device. Unlike typical device drivers, Simulink Real-Time device drivers:

- Can have driver code, that is C code written as an S-function using exported Simulink Real-Time kernel functions (see “Simulink Real-Time Exported Functions” on page 1-5).
- Can have an optional Simulink block interface (Simulink mask) that users use to configure the device and access output.
- Can have optional MATLAB code that you can write to perform operations such as cross-block checking or parameter value range checking. You reference this file through the Simulink mask.
- Can be included in a Simulink library.
- Can be configured like other Simulink blocks.



Anatomy of a Simulink Real-Time Driver

Before You Start

In this section...
“Introduction” on page 1-8
“Driver Types” on page 1-9
“Bus Types and Register Access” on page 1-9
“Register Access” on page 1-10
“Inlining Simulink Real-Time Drivers” on page 1-10

Introduction

This topic assumes that you satisfy the requirements outlined earlier in “Expected Background” on page 1-3 and that you have reviewed the following sections to prepare:

- “References” on page 1-4
- “Source Code” on page 1-5
- “Simulink Real-Time Exported Functions” on page 1-5
- “Third-Party Folder” on page 1-6

It also assumes that you already have a board for which you want to write a driver. Before you start, use the following checklist to specify the driver you want to write:

- Determine the functions of your board that you want to access with your driver.
- Determine the bus type for the board.
  - PCI
  - ISA
- Select the I/O access mapping type.
  - I/O port mapped
  - Memory address mapped
- Select polling versus interrupt.
- Specify the blocks for the drivers. Identify
  - Input and output ports
  - Mask parameters

    - Work variables to be shared between driver start, output, and terminate routines
- Determine your timing considerations.
- Decide whether you use Inlined functions.

  If yes, see the Target Language Compiler documentation of the Simulink Coder.

Driver Types

- Standard I/O
- Communication
- DMA
- Interrupt-driven

Bus Types and Register Access

The Simulink Real-Time software supports two standard bus types, ISA and PCI. The ISA bus is a 16-bit bus with an 8 MHz clock. Another form of ISA bus is the PC/104. The PCI bus is a 32-bit or 64-bit bus with a 33 MHz or 66 MHz clock. Another form of PCI bus is the PC/104+ (PC/104-Plus).

A driver performs I/O accesses through either I/O ports or memory addresses (memory mapped).

The Simulink Real-Time software accesses I/O port addresses for ISA and PCI buses as follows:

Bus	Access
ISA	Board switches or jumpers usually select I/O port address and memory-mapped regions.
PCI	The BIOS determines the I/O port address during PCI PNP (Plug and Play) configurations.

The memory space for I/O boards is different for ISA and PCI boards.

Bus	Memory Space
ISA	The Simulink Real-Time software only permits use of the memory address between 0xA0000 and 0xFFFFF

Bus	Memory Space
PCI	Upper memory address space, typically greater than 2 GB

Register Access

A device board supports either I/O port or memory-mapped access to onboard registers. See the board manufacturer's register programming documentation.

Inlining Simulink Real-Time Drivers

You can choose to inline or not inline Simulink Real-Time drivers. Note the distinction between Simulink and Simulink Coder conditional compilation. If you implement a device driver as an inlined S-function, the driver can coexist with Simulink Real-Time device drivers.

Inlining drivers allows you to customize code generated from Simulink Coder. If you choose to create inlined drivers, you must use the Simulink Coder Target Language Compiler.

Note: For convenience, you can create a noninlined version of the driver first, and create an inlined driver for the Target Language Compiler from the first driver.

Creating a Custom Driver

The following is a generic procedure for creating a custom device driver. See "Driver Authoring Tool Basics" on page 7-2 for a description of a tool that helps you create a simple custom driver that does not use DMA or interrupt handling.

You might need administrative or write privileges to add a custom device driver to the Simulink Real-Time system. Otherwise, see "Block Creation". This topic describes how to add custom blocks to a library.

1. Write your driver in C using the approved I/O functions. An example device driver for the analog inputs of the Diamond MM-32 board is available at

   matlabroot\toolbox\rtw\targets\xpc\target\build\xpcblocks\addiamondmm32.c

2. As you write your device driver, you will want to compile and link the driver, then test it. Compile and link the driver into a MEX-file. For example:

   ```
   mex driver.c
   ```

   This step creates the MEX-file executable, *driver*.mexw32.

   A MEX-file is used for simulation on the development computer and to set data structure sizes during code generation. It is not used during execution on the target computer.

3. Create a file of MATLAB code to supplement the main C driver and support the block mask. For an example of this file, see

   matlabroot\toolbox\rtw\targets\xpc\target\build\xpcblocks\maddiamondmm32.m

4. Open the Simulink Library Browser and create a new library, for example, *your_company_name*lib (see "Create Block Libraries"). Use a unique library name to prevent conflicts with other libraries.

5. In the new library, create an S-function block by dragging it from the Simulink Library Browser.

6. To configure the S-Function block, in the new library, right-click the S-Function block and select **Block Parameters (S-function)**.

7. In **S-function name**, enter the name, without extension, of the driver. For example, addiamondmm32, the name of the example driver.

8. In **S-function parameters**, enter the parameters you defined for the driver. The parameter names you enter here must match the names you enter through the **Parameters & Dialog** and **Initialization** panes of the Mask Editor dialog box. For

example, enter `configuration`, `firstChan`, `numChans`, `range`, `sampleTime`, and `baseDec`.

9. Leave **S-function modules** set to the default value, unless you need to separate your driver C file into multiple files. If so, see "Specify Additional Source Files for an S-Function".
10. Double-click the S-Function block and create a block mask (see "Driver Mask Guidelines" on page 4-3).
11. Save and close the S-Function block.
12. At the bottom of the S-Function block, enter a block name. For example, `MM-32`.
13. Save and close the library.
14. To make your new library visible in the Simulink Library Browser, move it to

    matlabroot`\toolbox\rtw\targets\xpc\target\build\xpcblocks\thirdpartydrivers`
15. Copy and paste `sample_xpcblocks.m` in

    matlabroot`\toolbox\rtw\targets\xpc\target\build\xpcblocks\thirdpartydrivers`

    Rename this file *`your_company_name`*`lib_xpcblocks.m` and edit this file as follows:

    - Set `out.Library` to your new library.

      ```
      out.Library = 'your_company_namelib';
      ```
    - Set `out.Name` to a string, such as the library name.

      ```
      out.Name = 'your_company_namelib Blockset';
      ```

      This string will appear in the Simulink Library Browser.
    - Set `out.IsFlat` to `0`.

      ```
      out.IsFlat = 0;
      ```

    Create a function that calls the `out` structure.
16. (Optional for PCI boards) To enable the `SimulinkRealTime.Target.getPCIInfo` function access your new board, copy `sample_supported.m` to a unique file name. For example:

    `your_company_name``lib_supported.m`

    Edit your copy of the file. For each board for which you add a device driver:

a Copy one of the commented structures in the file.

b Remove the comment symbols (%).

c Starting with 1, update the ID number.

Tip Number the device structures sequentially, starting with 1.

d Replace the field entries with your equivalents.

A structure entry might look like:

```
boards(1).VendorID   = '18f7';
boards(1).DeviceID   = '0004';
boards(1).SubVendorID = '-1';
boards(1).SubDeviceID = '-1';
boards(1).DeviceName = '422/2-PCI-335';
boards(1).VendorName = 'Commtech';
boards(1).DeviceType = 'Serial Ports';
boards(1).ADChan = '0';
boards(1).DAChan = '0';
boards(1).DIOChan = '0';
boards(1).Release = 'R2007a';
boards(1).Notes = 'Support any baud rate from 1 bps
     to 6.25 Mbps';
```

e Save and close the file.

f To confirm your entries, in the Command Window, type:

```
tg = slrt;
getPCIInfo(tg, 'all')
```

17 Place all your driver files, including the include files, in the folder:

```
matlabroot\toolbox\rtw\targets\xpc\target\build\xpcblocks\thirdpartydrivers
```

To prevent conflicts with the driver files for Simulink Real-Time blocks and third-party drivers, give all driver files unique names.

18 To update the folders that you added, at the MATLAB Command Window, type

```
rehash toolbox
```

When you are done, your library appears in the Simulink Library Browser with `Simulink Real-Time:` added to the beginning of the library name.

Simulink Real-Time: *your_company_name*lib Blockset

Debugging Notes

While developing your custom driver, you can use `printf` statements in your code. This displays output in the output area of the target computer monitor. If your `printf` statements scroll off the monitor, consider booting the target computer in "Text" mode. This will disable graphics on the target computer display and remove the entire scope area to provide more display area for printf statements.

1. At the MATLAB Command Window, type `slrtexplr` to start Simulink Real-Time Explorer.
2. In the Targets pane, expand **MATLAB Session** and expand the target computer.
3. Click **Properties** or click the Properties button in the toolbar.
4. Click **Target settings** and clear the **Graphics mode** check box.
5. Recreate the target boot disk and reboot the target computer.

   The scope area on the target computer monitor does not appear.

   Tip Target scopes are automatically converted to host scopes, including target scopes dynamically added during execution.
6. Continue with device driver development.

2

PCI Drivers

- "PCI Bus Considerations" on page 2-2
- "Sample PCI Device Driver" on page 2-7

PCI Bus Considerations

In this section...
"Introduction" on page 2-2
"PCI Configuration Space API" on page 2-3
"Memory-Mapped Accesses" on page 2-5
"I/O Port Accesses" on page 2-6

Introduction

When writing Simulink Real-Time drivers for PCI devices, consider the memory access method. A PCI device can be either I/O port mapped or memory mapped.

- I/O port mapped — The BIOS assigns a port range.
- Memory mapped — The BIOS assigns a memory region, if your device is memory mapped.

The computer BIOS automatically assigns a conflict-free set of resources to the PCI devices found in the system at boot-up. You typically do not know where the board resides (base address) before driver initialization. However, you can obtain this information by querying the PCI configuration space at run time. The Simulink Real-Time software provides functions to accomplish this.

To locate a PCI device, you need the following:

- Vendor and device ID
- Optionally, subsystem vendor and subsystem device ID

  Note: You need the subsystem vendor and subsystem device ID if the vendor and device ID do not uniquely identify the board.

- Slot number or bus and slot number

You can have the drivers locate PCI devices in one of the following ways:

- If the system has one board of a given type, you can use the driver `slot` option to search for the first board that matches a vendor and device ID. To initiate this search, set this option to `-1`.

- If the system contains multiple boards of the same type, setting the `slot` option to `-1` does not find the additional boards. In that case, specify the bus and slot numbers with the vendor and device IDs.

PCI Configuration Space API

Before you can access a PCI device, you need to access the configuration space to locate the board in the target computer memory. This section describes the procedure to do this.

For PCI devices, the driver will need to access the PCI configuration space for the board. This space contains relevant board information such as the base address and access type (I/O port or memory mapped). The Simulink Real-Time software provides functions that allow the driver to access this space.

- Vendor and device ID — The driver searches the boards for the specified vendor (manufacturer) and device ID. The PCI Steering Committee, an independent standards body, assigns a unique vendor ID (`uint16`) to each PCI board vendor. Each vendor then assigns a unique ID to each PCI board type it supports.

  Note: Vendor and device IDs might not uniquely identify a board. For example, boards that use the PLX-9080 bus interface chip have a vendor ID of 10B5 (the vendor ID assigned to PLX Technology, Inc.). The device ID for the chip is 9080. In cases like this, to select a particular board that contains this chip, you must use a subvendor and subdevice ID in addition to the vendor and device IDs.

- Slot number or bus and slot number — The driver looks only for the board that matches the specified vendor and device ID and slot number.

PCI Device Information

Use the `xpcGetPCIDeviceInfo` function to get information for a PCI device in your system. The syntax for this function is:

```
int xpcGetPCIDeviceInfo (uint16_T vendorId, uint16_T deviceId,
uint16_T subVendorId, uint16_T subDeviceId, uint16_T bus, uint16_T
slot, xpcPCIDevice *pciInfo);
```

This function returns the `xpcPCIDevice` structure filled according to the following:

If You Supply...	This Function....
All four parameters	Looks for a device that matches all four parameters and returns the `xpcPCIDevice` structure for that device. Use this form if you know that your system has multiple boards from the same vendor with the same ID and you want your user to specify the exact device.
`XPC_NO_SUB` for the *`subVendorId`* or *`subDeviceId`* parameter	Does not consider the subvendor or subdevice ID when looking for a match for the specified device. It returns the `xpcPCIDevice` structure for a device that matches the other parameters. You can use this form if you do not plan to use the *`subVendorId`* or *`subDeviceId`* values.
`XPC_NO_BUS_SLOT` for the slot for the device	Returns the first PCI device it finds that matches the remaining parameters. You can use this form if you know that your system has only one board with a particular ID set.

Passing Slot Information from the Block Mask to Its Driver

Simulink Real-Time drivers use the following convention to fill in slot parameters and retrieve slot information. Choose the convention that will work best for you.

Set...	To...
Set `slot = -1`	Assume `bus = 0` and call the `xpcGetPCIDeviceInfo` function to find the first instance of the board.
Set `slot = S`	Assume `bus = 0` and call the `xpcGetPCIDeviceInfo` function to find the specified board. If the board matches the IDs, return the PCI information to the driver. Otherwise, return an error.
Set `slot = [B, S]`	Check bus *`B`* and slot *`S`* for the specified board. If the board matches the IDs, return the PCI information to the driver. Otherwise, return an error. Setting `slot = [0, S]` is identical to `slot = S`.

The following example illustrates how to use the `xpcGetPCIDeviceInfo` function to program the driver to accept slot number input or slot and bus number input from the driver block.

1 Call this function from the `mdlStart` callback function.

2 Pass the slot number or slot and bus number into the `xpcGetPCIDeviceInfo` function using code like the following:

```
uint16_T       vendorId, deviceId;
int32_T        bus, slot, subvendor, subdevice;
xpcPCIDevice   pciInfo;

/* S_PCI_SLOT_ARG is passed in from the mask */
/* Typically the slot arg is a scalar containing -1 if the target
has only one board of this type */
/* If the target has multiple boards of this typem, the slot arg
is a vector containing bus and slot info */
/* This code snipped parses the slot arg into bus and slot */
if ( (int_T)(mxGetN(ssGetSFcnParam(S, S_PCI_SLOT_ARG))) == 1 ) {
 bus  = 0;
 slot = (int32_T)(mxGetPr(ssGetSFcnParam(S, S_PCI_SLOT_ARG))[0]);
} else {
 bus  = (int32_T)(mxGetPr(ssGetSFcnParam(S, S_PCI_SLOT_ARG))[0]);
 slot = (int32_T)(mxGetPr(ssGetSFcnParam(S, S_PCI_SLOT_ARG))[1]);
}

vendorId = (uint16_T)0x1234;
deviceId = (uint16_T)0x9876;
subvendor = (uint16_T)0x5678;
subdevice = (uint16_T)0x8765;
/* Set subvendor and subdevice to XPC_NO_SUB, XPC_NO_SUB if they are not required */

/* xpcGetPCIDeviceInfo() populates the pciInfo struct */
if ( xpcGetPCIDeviceInfo(vendorId, deviceId,
     subvendor, subdevice,
     bus, slot,
     &pciInfo) ) {
 sprintf(msg, "Board 0x%x not found at bus %d slot %d", deviceId, bus, slot);
 ssSetErrorStatus(S, msg);
 return;
}
```

For detailed information on the `xpcPCIDevice` structure, see `xpcPCIDevice`.

Memory-Mapped Accesses

A memory-mapped PCI board uses up to six memory regions to access board regions and memory. Each region might also have a different length. You must call the `xpcReserveMemoryRegion` function for each PCI memory region you want to access; use the returned virtual address to access the region. Failure to do this will result in a segmentation fault.

To access a memory mapped location, do the following:

1 Declare a variable of the required pointer type to hold the memory location. For example:

```
volatile uint32 *csr; /* Control and status register */
```

Note: Use the `volatile` keyword here; otherwise, the compiler might optimize away accesses to this location.

2 Set the pointer value (address) to the physical address at which the register resides.

I/O Port Accesses

To access I/O ports, use the following functions:

- `xpcInpB`, `xpcInpW`, `xpcInpDW` — I/O port input functions for byte, word, and double word accesses
- `xpcOutpB`, `xpcOutpW`, `xpcOutpDW` — I/O port output functions for byte, word, and double word accesses

Sample PCI Device Driver

To access example PCI device driver code, type:

```
cd(fullfile(matlabroot,'toolbox','rtw','targets','xpc','target','build','xpcblocks'));
```

This folder contains driver code for boards supported by Simulink Real-Time.

Note: To specify the S-function name for an S-Function block, enter the C file name without the extension.

3

ISA and PC/104 Drivers

ISA and PC/104 Bus Considerations

In this section...
“Introduction” on page 3-2
“I/O Mapped” on page 3-2
“Memory Mapped” on page 3-2

Introduction

When writing Simulink Real-Time drivers for ISA and PC/104 devices, consider the memory access method. A PCI device can be either port mapped or memory mapped. Most ISA and PC/104 boards are port mapped. Those that are memory mapped typically need large register banks or are interfaced via dual-port memory.

Note: The Simulink Real-Time kernel does not support ISA and PC/104 PNP boards. This means that you can write Simulink Real-Time device drivers only for ISA and PC/104 boards for which you can set the base address manually. To manually set the base address, insert jumpers or move DIP switches on the board.

- Port mapped

  The base port address on the board is set via jumpers or switches. You might need to reset these addresses to resolve conflicts.
- Memory mapped

  The I/O and memory on the board are set via jumpers or switches. You might need to reset these addresses to resolve conflicts.

I/O Mapped

The base port address on the board is set via jumpers or switches. Drivers cannot discover these addresses on their own; you must specify these addresses to the driver.

Memory Mapped

The I/O and memory on the board is set via jumpers or switches. Drivers cannot discover these addresses on their own; you must specify these addresses to the driver.

Reserved Space on the Target Computer

The Simulink Real-Time kernel reserves space in the region (C0000 to DC000) for memory-mapped I/O cards. You must set up ISA and PC/104 cards to use addresses in this range.

Masking Drivers

- "Creating Driver Subsystem Masks" on page 4-2
- "Driver Mask Guidelines" on page 4-3
- "Cross-Block Checking" on page 4-5
- "When You Are Done" on page 4-6
- "Sample Driver Mask" on page 4-7
- "Configure Parameters & Dialog Pane" on page 4-8
- "Configure Initialization Pane" on page 4-10
- "Configure Documentation Pane" on page 4-11
- "Create Initialization Callback" on page 4-12

Creating Driver Subsystem Masks

This topic describes guidelines for creating a Simulink block user interface (mask) for the S-Function block associated with your driver. A mask defines the menu items that will be passed to the S-function. The mask can call a MATLAB file to do parameter or range value checking. You can also modify the labels of a block to show port numbers or other information. After you create the C code for a Simulink Real-Time device driver:

1. Create an optional MATLAB file.
2. Create an S-Function block for the driver.
3. Create a mask for the S-Function block.

   This is the basic Simulink mask, with parameters and descriptions as required by the block. When you are done, you can make the device driver and its mask available for users to add to their models.

Driver Mask Guidelines

This topic lists guidelines you should follow when creating a mask for your Simulink Real-Time driver. (See “Mask a Block” and “Best Practices for Masking”.)

Users access the masked block to interact with the driver, which in turn interacts with the device.

- Create an S-Function block for the driver.
- Decide on the set of parameters the user will need to provide to the driver. You should have already programmed this into the driver C code itself.
- Select descriptive names for these parameters.

  For ease of use, choose parameter names that are as close as possible to those used by Simulink Real-Time drivers.
- For each parameter, decide if the parameter can accept a finite number of possible input values. If yes, consider using one of the following widgets:

  - Check box — For yes/no or 1/0 inputs
  - Drop-down list — For a finite list of choices

  Your mask can also be dynamic, where the dialog changes according to user selections.
- Choose descriptive variable names.
- Configure the library block so that the block mask modifies its label according to user input. For example, a check box might cause the dialog to change.
- Terminate the title beneath the driver block with a blank space. This is because if a model contains more than one block of a given type, Simulink appends a number to the title under the block. Adding a blank space to the end of the label improves readability.
- Name the block so that it indicates the purpose of the driver.
- If you want to link help information to the mask **Help** button, see “Create Mask Documentation”.
- From within the mask, you can call a custom file to perform a number of operations, including the following:

  - Range checking for all parameters. For example, if you expect input values from 1 to 10, do not allow users to enter negative values, or values greater than 10.

- Cross-block checking (see “Cross-Block Checking” on page 4-5).

Cross-Block Checking

Cross-block checking determines if multiple blocks are trying to access the same I/O module. You should include cross-block checking in your driver to prevent such conflicts. You can perform cross-block checking by calling `find_system` from the block mask in a number of ways. Use the following guidelines when performing cross-block checking:

- You should call the `find_system` function from the block `InitFcn` callback function. There are two phases of MATLAB file execution during an update system operation. If you call the `find_system` function from a block `InitFcn` callback function, defined in the Block Parameters dialog of the block, no additional updates are triggered.
- Decide on the level of cross-block checking for your I/O configuration. For example, boards that use the 8255 chip for digital I/O need to check if two different blocks are requesting opposing directions (for example, input and output) for the same group of 8 bits. On this chip, there are three groups of 8 bits. You can configure each group for input or output. The associated Simulink Real-Time driver generates an error in `InitFcn` if `find_system` detects that two blocks are trying to use the same group of 8 bits for input and output. See

  ```
  matlabroot\toolbox\rtw\targets\xpc\target\build\xpcblocks\mpci8255.m
  ```

  which is called as `mpci8255(1)` for the Measurement Computing™ PCI-DAS 1200 digital input and output blocks. During an update diagram sequence, Simulink calls the `InitFcn` callback function once for each block. Simulink might call the initialization commands in the mask multiple times.

When You Are Done

After you write the driver S-function and create the S-Function block, optional block mask, and MATLAB file for it, be sure to:

- Check the text of each error message for clarity and spelling.
- (Optional) Use a coding standard indentation such as four or eight spaces without tabs.
- Copy your new blocks into a custom folder with a unique name.

  To enable your new blocks to be viewable in the Simulink Library Browser, see "Creating a Custom Driver" on page 1-11.
- Test the driver for the following:
  - Run the `mex` command on the driver to build the driver for simulation and code generation.
  - Verify the I/O module behavior under as many conditions as possible.

Sample Driver Mask

This example procedure recreates the block mask for the Diamond MM-32 Analog Input block.

1. To create the initial block mask, right click on the Diamond MM-32 Analog Input block and select **Mask > Create Mask**.
2. “Configure Parameters & Dialog Pane” on page 4-8
3. “Configure Initialization Pane” on page 4-10
4. “Configure Documentation Pane” on page 4-11

Configure Parameters & Dialog Pane

The Diamond MM-32 Analog Input block has the following user-visible parameters.

Variable name	Type	Prompt	Evaluate	Tunable
`configuration`	Popup	`Channel configuration`	Yes	Yes
`firstChan`	Edit	`First channel number:`	Yes	Yes
`numChans`	Edit	`Number of channels:`	Yes	Yes
`range`	Popup	`Range`	Yes	Yes
`sampleTime`	Edit	`Sample time:`	Yes	Yes
`base`	Edit	`Base address (for example 0x300):`	No	Yes

Configure the **Parameters & Dialog** pane of the block mask using the following procedure.

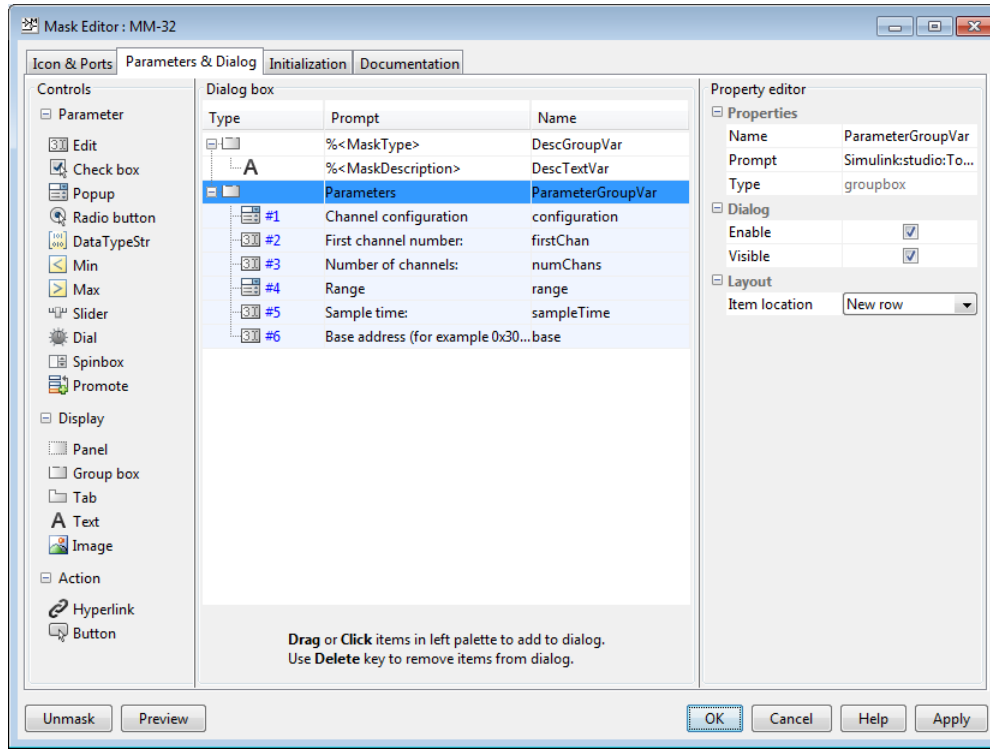
1. Select the **Parameters & Dialog** tab.
2. To enter the parameter for variable `configuration`, double-click the Popup parameter icon under **Parameters**.

   A **groupbox** node representing the popup parameter appears. The first time a parameter is added, **groupbox** node representing the mask also appears.
3. In the **Prompt** column, enter `Channel configuration`.
4. In the **Name** column, enter `configuration`.
5. Under the **Attributes** node to the right, verify that **Evaluate** and **Tunable** are both selected.
6. To enter the parameter for variable `firstChan`, double-click the Edit parameter icon under **Parameters**.
7. In the **Prompt** column, enter `First channel number:`.
8. In the **Name** column, enter `firstChan`.
9. Under the **Attributes** node to the right, verify that **Evaluate** and **Tunable** are both selected.

10 Enter parameters for variables `numChans`, `range`, `sampleTime`, and `base` using the same procedures.

Clear the **Evaluate** check box for variable `base`.

After you are finished, the Mask Editor **Parameters & Dialog** pane looks like this:



11 Click **OK** to save the mask.

The next step is “Configure Initialization Pane” on page 4-10.

Configure Initialization Pane

1. Select the **Initialization** tab. The pane displays the dialog variables you entered in the **Parameters & Dialog** pane.
2. In the **Initialization commands** code section, type:

   ```
   phase = 2;
   [baseDec, maskDisplay, maskDescription] =
   maddiamondmm32(phase, configuration, firstChan,
   numChans, range, base);
   set_param(gcb, 'MaskDisplay', maskDisplay);
   ```

   where `maddiamondmm32` references the `maddiamondmm32.m` file for the driver.
3. Create the `InitFcn` callback for the block. A model calls this callback at the start of model simulation. See "Create Initialization Callback" on page 4-12.
4. Click **OK** to save the mask.

The next step is "Configure Documentation Pane" on page 4-11.

Configure Documentation Pane

1. Select the **Documentation** tab. This tab contains three fields, **Mask type**, **Mask description**, and **Mask help**.
2. In the **Mask type** field, enter the type of driver. For example:

   ```
   addiamondmm32
   ```

3. In the **Mask description** field, enter a description for the driver, . For example:

   ```
   MM-32
   Diamond
   Analog Input
   ```

4. In the **Mask help** field, if you are providing online documentation to associate with the **Help** button, call that online documentation from this field. See “Create Mask Documentation”.
5. Click **OK** to save the mask.

You can also enter the mask description in the **Initialization** code section using a command such as:

```
set_param(gcb, 'MaskDescription', maskDescription);
```

If you enter the mask description in the **Initialization** section, omit entering it in the **Mask description** field.

Create Initialization Callback

After you create the block mask, define an `InitFcn` callback for the block. A model calls this callback at the start of model simulation.

1. Right-click the block and select **Properties** from the drop-down list.
2. Select the **Callback** tab from the dialog. From the list, select `InitFcn`. Enter MATLAB code (usually a function call) to perform initialization-time-only tasks, such as categorizing I/O ports or doing cross-block error checking.

   Initialization-time tasks have special requirements, such as:

   - Gathering consistent information about the block inputs and outputs. For example, when Simulink is categorizing digital I/O ports as input or output, both the digital input and the digital output `InitFcn` must return the same list or Simulink may misconfigure the block.
   - Doing cross-block error checking using `findsystem`. The function `findsystem` should only be called at `InitFcn` time. If called at mask initialization time, `findsystem` could force multiple reevaluations of the whole model.

   For ease in debugging, the MATLAB code should be a single call to a MATLAB initialization function `InitFcn`. `InitFcn` could be implemented as a separate function, but it is sometimes convenient to combine `InitFcn` with a mask initialization function `MaskInit` (in this case, `maddiamondmm32.m`). To do this, write `MaskInit` to be called at `InitFcn` call time:

   - Save a derived configuration as `UserData` on the block and retrieve it again during the mask initialization call itself. The `InitFcn` call-time code cannot return a value to the mask.
   - Omit passing the mask parameter variables into `MaskInit` from the `InitFcn` dialog box. Mask parameter variables are not defined at `InitFcn` call time.
   - Isolate the `InitFcn` call-time code from code that uses an omitted mask parameter value. (If the execution path references an omitted parameter value, MATLAB will raise an error.) Use a method such as the following to isolate the `InitFcn` code:

     - Pass a single argument of arbitrary value, then use `nargin` to determine the number of parameters:

       ```
       MaskInit(Arg1, Arg2, Arg3, ..., ArgN)
       if (1 == nargin)
       ```

```
    % Initialization code, which must not use Arg2, Arg3, ..., ArgN
else
    % All other processing
end
```

- Pass a single argument of value 1, then check for that value using a switch on the first parameter:

```
MaskInit(Arg1, Arg2, Arg3, ..., ArgN)
switch (Arg1)
case 1:
    % Initialization code, which must not use Arg2, Arg3, ..., ArgN
otherwise
    % All other processing
end
```

- To get mask parameter values, call `get_param( gcb, 'paramvariablename' )` to get their ASCII value. For example:

```
function [baseDec, maskDisplay, maskDescription] = ...
        maddiamondmm32(phase, configuration, firstChan, numChans, range, base) %#ok

    vendorName   = 'Diamond';
    deviceName   = 'MM-32';
    description  = 'Analog Input';
    maskType     = 'addiamondmm32';

    if phase ~= 2  % assume InitFcn unless phase 2
        base = get_param( gcb, 'base' );
        blocks = find_system(bdroot, 'FollowLinks', 'on', ...
        'LookUnderMasks', 'all', 'MaskType', maskType, 'base', base);
        if length(blocks) > 1
            error(Simulink Real-Time:DiamondMM32:Block',...
                  'Only one Diamond Systems MM-32 A/D block per ...
                   physical board allowed in a model - each block of ...
                   this type must have a distinct ISA address.');
        end
        return
    end
```

- In this file, you should check the range values of the parameters. Checking the mask here will catch illegal values early in the build process.
- This example returns a string to display on the block with the variable `port_label` commands with which to label the input and output ports. The number and content of the `port_label` commands depend on the `channel vector` that the user enters in the mask.

Interrupt Support

- "Use Interrupts in Simulink Real-Time" on page 5-2
- "Add Interrupt Support" on page 5-5
- "Hook Function Prototypes — Alphabetical List" on page 5-12

Use Interrupts in Simulink Real-Time

If your device supports interrupts, you can use these procedures to add your custom interrupt functions to the Simulink Real-Time framework.

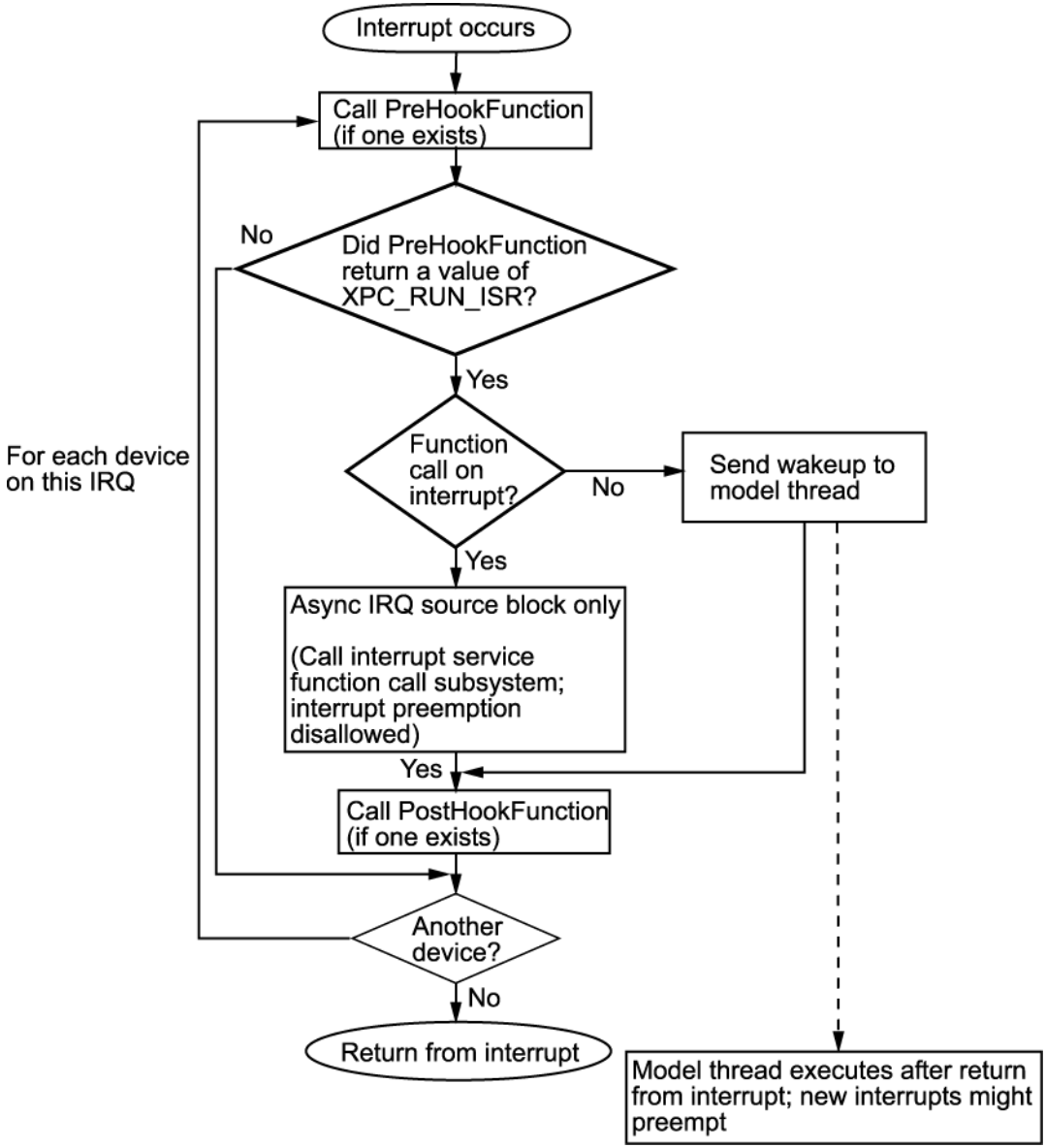
Your users can use interrupts in real-time applications in one of the following ways:

- Use the interrupt with the Simulink Real-Time `Async IRQ Source` block to execute a function-call subsystem when an interrupt occurs.
- Use the interrupt to run the model in place of the timer interrupt, available through the model Configuration Parameters dialog box in the **Code Generation** > **Simulink Real-Time Options** pane.

Note: Although users can use interrupts in two ways, you program for these interrupts using the same procedure. See "Add Interrupt Support" on page 5-5.

When a model executes, it executes in the following order:

1. Call all `mdlStart` routines in block execution order.
2. Call the interrupt `Start` function, if one exists.
3. Allow background graphics and network tasks to run until an interrupt occurs.


Interrupt occurs
Call PreHookFunction (if one exists)
No
Did PreHookFunction return a value of XPC_RUN_ISR?
Yes
For each device on this IRQ
Function call on interrupt?
No
Send wakeup to model thread
Yes
Async IRQ source block only
(Call interrupt service function call subsystem; interrupt preemption disallowed)
Yes
Call PostHookFunction (if one exists)
Another device?
No
Return from interrupt
Model thread executes after return from interrupt; new interrupts might preempt

When a device interrupt occurs, the generated code uses the following steps for each device on this IRQ to determine which device generated the interrupt:

1. Call the `PreHook` function, if one exists. The return value determines the action.
2. The generated code determines whether this device generated the interrupt.
   - If the `PreHook` function returns `XPC_RUN_ISR`, execution continues to step 3.
   - If the `PreHook` function returns `XPC_DROP_ISR`, the generated code goes to step 5.
3. The generated code determines whether the Async IRQ Source block has a function-call subsystem connected.
   - If so, the generated code calls the interrupt service function-call subsystem. When the interrupt service function subsystem returns, the generated code goes to step 4.
   - If this board is configured to run the model, and this board did interrupt, send a wake-up call to the model thread. The model thread does not immediately execute. It waits until after the boards that use this IRQ have been checked and the return from the interrupt has been executed.
4. Call the `PostHook` function, if one exists. If one does not exist, the generated code goes to step 5.
5. The generated code checks whether another device using the same IRQ exists in the system.

   If so, execution returns to the beginning of step 1.

   Otherwise, generated code goes to step 6.
6. Returns from the interrupt.

The Simulink Real-Time kernel now runs the highest priority thread. The highest priority ready thread is the model if a wake-up call was sent to it.

Add Interrupt Support

In this section...
"Introduction" on page 5-5 "Guidelines for Creating Interrupt Functions" on page 5-7 "Filling in the Driver `board` Structure" on page 5-8

Introduction

To add interrupt handling for a custom driver, you must create

- A descriptor file to connect a board type to the functions to start, handle, and stop interrupts
- A C file to implement these functions

Include the following functions. See "Hook Function Prototypes — Alphabetical List" on page 5-12 for the prototype details.

Function	Description
`PreHook`	Runs just before either a function-call subsystem or entire model is called. Program this function to acknowledge the interrupt and cause the board to stop issuing the interrupt signal.
`PostHook`	Runs after return from function call on interrupt, and before model execution. It is typically not used.
`Start`	Runs as the last item when starting a model, just before the model runs. It is typically used to turn on interrupt generation. Program this function to enable interrupts on the board and start any action.
`Stop`	Runs at the beginning of a stop request, before any `mdlTerminate` entries for any block in the model runs. It is typically used to turn off interrupt generation. Program this function to disable interrupts from the board and stop any action. This is the first action called, when a real-time application stops executing.

Note: You must use the `Stop` function to turn off interrupts if you have turned them on in the `Start` function. In this way, the stop and start functions should cancel each other.

To add interrupts for your custom driver, use the following general steps:

1. Create a hook file in the following folder:

   ```
   matlabroot\toolbox\rtw\targets\xpc\target\build\
   xpcblocks\thirdpartydrivers
   ```

   Hook files are C files (`.c`). For example, look at files in *`matlabroot`*`\toolbox\rtw\targets\xpc\target\build\src`, such as `xpc6804hooks.c`.

2. Name the hook file something like:

   `your_company_name_board``_hook.c`

3. In the hook file, create the interrupt functions `PreHook`, `PostHook`, `Start`, and `Stop`. See "Guidelines for Creating Interrupt Functions" on page 5-7 for information on how to create these functions.

4. Copy the file *`sample`*`_int.m` to a unique file name in the following folder:

   ```
   matlabroot\toolbox\rtw\targets\xpc\target\build\
   xpcblocks\thirdpartydrivers
   ```

   For example:

   `your_company_name``_int.m`

   The Simulink Real-Time software searches in this folder for file names that end with `_int.m` and looks for board interrupt descriptions.

5. Open and edit the following file:

   ```
   matlabroot\toolbox\rtw\targets\xpc\target\build\
   xpcblocks\thirdpartydrivers\your_company_name_int.m
   ```

   Add to this file a `board` structure for each Simulink Real-Time supported board for which interrupt functions have been written. See "Filling in the Driver `board` Structure" on page 5-8 for a description of how to fill in a `board` structure.

6. Save and close the file.

7. At the MATLAB Command Window, type:

   ```
   rehash toolbox
   ```

8 Restart the MATLAB interface to update the Async IRQ Source block and Configuration Parameters dialogs.

Guidelines for Creating Interrupt Functions

Simulink Real-Time interrupt functions have predefined purposes and typically follow a particular order. This section describes the guidelines on creating interrupt functions. See "Hook Function Prototypes — Alphabetical List" on page 5-12 for the prototypes for these functions.

To prepare for the creation of the hook file, examine the existing Simulink Real-Time hook files (*matlabroot*`\toolbox\rtw\targets\xpc\target\build\src`) and copy and modify one that is the same board type, PCI or ISA, as the board for which you are creating a custom driver. For example, `xpc6804hooks.c` is for an ISA board. Place your new file in

```
matlabroot\toolbox\rtw\targets\xpc\target\build\
xpcblocks\thirdpartydrivers\
```

When modifying an existing hook file:

- Change the names of the functions to match those you have selected for your board.
- Do not change the function signatures.
- Do not remove the `__cdecl` string.
- The `PreHook` and `PostHook` functions run with interrupts disabled. Do not change the interrupt status in these functions.

When writing the interrupt functions, note the following:

- When an interrupt occurs, the kernel calls the `PreHook` function.

  Note: This function is run with interrupts disabled. If this function cannot turn off the interrupt, an infinite loop will occur because the interrupt service routine (ISR) will continuously call the `PreHook` function.

- Because the `PostHook` function has limited use, you most likely do not need to define this function. Set this function to `'NULL'` if you do not need it.
- The generated code calls the `Start` function during the startup phase of model execution as the last action, after the model has called all the `mdlStart` routines.

This function is typically used to enable interrupts from the board. The real-time application is ready to accept interrupts a few microseconds after this function is called. Do not try to enable interrupts from the board `mdlStart` function.

- When a real-time application stops executing, the generated code calls the `Stop` function first. Disable interrupts from the board in this function.

Filling in the Driver `board` Structure

This section describes how to fill in a driver `board` structure, element by element.

- Depending on the bus type of your board, select a `board` structure of an existing board that has the same bus type. The information passed to the functions is slightly different for an ISA board or a PCI board. You will use this structure as a template for your own board entry. The following is a structure for an ISA or PC/104 device:

```
board.name = 'RTD_DM6804';
board.VendorId = -1;
board.DeviceId = 1;
board.SubVendorId = -1;
board.SubDeviceId = -1;
board.PreHookFunction = 'xpc6804';
board.PostHookFunction = 'NULL';
board.HookIncludeFile = 'xpc6804hooks';
board.StartFunction = 'xpc6804start';
board.StopFunction = 'xpc6804stop';
```

  The following is a structure for a PCI device:

```
board.name = 'General Standards 24DSI12';
board.VendorId = hex2dec('10b5');
board.DeviceId = hex2dec('9080');
board.SubVendorId = hex2dec('10b5');
board.SubDeviceId = hex2dec('3100');
board.PreHookFunction = 'xpcgs24dsi12prehook';
board.PostHookFunction = 'NULL';
board.HookIncludeFile = 'xpcgs24dsi12hooks';
board.StartFunction = 'xpcgs24dsi12start';
board.StopFunction = 'xpcgs24dsi12stop';
```

- `name` — Enter a name string for the board. The Simulink Real-Time software uses this string to populate the drop-down list for the **I/O board generating the interrupt** parameter in the following:

  - Async IRQ Source block
  - **PCI slot (-1: autosearch) or ISA base address** parameter in the **Simulink Real-Time Options** section of the model Configuration Parameters dialog box

- `VendorId`, `DeviceId`, `SubVendorId`, `SubDeviceId` — Enter the ID strings for the board. If you have a PCI board, the board manufacturer identifies that board with either two or four ID values, depending on the specific board. When calling the hook functions, the Simulink Real-Time kernel obtains the PCI information for the board and passes it to the hook functions. Use these parameters to help identify the interrupting board.

  - For `VendorId` and `DeviceId`, enter the IDs you get from the board manufacturer.
  - Many boards do not have `SubVendorId` and `SubDeviceId` values. In these cases, insert the value `-1` to prevent The Simulink Real-Time software from checking for them.

  If you have an ISA board, it does not have a vendor or device ID; instead, the generated code will insert the ISA base address in the first base address entry of the PCI structure. To indicate to the kernel that this is an ISA board, set `VendorId` to `-1` and `DeviceId` to `1`.

  If you do not need hook functions:

  - Set `VendorId` to `-1` and `DeviceId` to `-1`.
  - Set `Fnc` and `PostHookFcn` to `'NULL'`.
  - Set `StartFunction` and `StopFunction` to `'NULL'`.

  The Async IRQ Source block will still call the subsystem when an interrupt occurs.

  The following table summarizes your options for this element:

VendorId	DeviceId	Usage
+ID	+ID	PCI board
-1	+1	ISA board
-1	-1	Special case: If the driver does not need hook functions. The driver can still use the Async IRQ Source block. As an example, see the source code for the serial driver.

- Enter the names of the interrupt functions. See “Hook Function Prototypes — Alphabetical List” on page 5-12 for the prototype details.

  - `PreHookFunction`

    Prototype:

    ```
    int __cdecl  your_company_name_boardPreHook(xpcPCIDevice *pciInfo);
    ```

  - `PostHookFunction`

    Prototype:

    ```
    void __cdecl your_company_name_boardPostHook(xpcPCIDevice *pciInfo);
    ```

  - `StartFunction`

    Prototype:

    ```
    void __cdecl your_company_name_boardStart(xpcPCIDevice *pciInfo);
    ```

  - `StopFunction`

    Prototype:

    ```
    void __cdecl your_company_name_boardStop(xpcPCIDevice *pciInfo);
    ```

  If any of these four functions does not need to exist, set the corresponding `board` structure entry to `'NULL'` to prevent calls to that function in that context.

Note: The differences between hook functions for PCI and ISA devices are:

- PCI devices — For a PCI device, the driver must fill in the *`VirtAddress`* field of the `xpcPCIDevice` structure in the `Start` function before the hook function is called. In the Start function, call the `xpcReserveMemoryRegion` function and save the resulting virtual address in the *`VirtAddress`* field. The remaining fields of the structure are filled in automatically, A pointer to the same instance of this structure is passed to all four functions.
- ISA devices — For an ISA device, the base I/O address is entered in the first physical address of the `xpcPCIDevice` structure. No other fields in the structure are filled in. The hook function must complete the structure.

- `HookIncludeFile` — Interrupt handling file that contains the `PreHookFunction`, `PostHookFunction`, `StartFunction`, and `StopFunction` functions for this board. Specify this name without the `.c` extension.
- Specify this structure for each board for which interrupt functions have been written. For example:

```
board(1).name = 'name1';
.
.
.
board(2).name = 'name2';
```

Hook Function Prototypes — Alphabetical List

PostHook

Run after return from interrupt service routine function-call subsystem or after sending wake-up call to model thread

Syntax

```
void __cdecl your_company_name_boardPostHook(xpcPCIDevice *pciInfo);
```

Argument

pciInfo Pointer to the `PciDevice` structure.

Description

The *your_company_name_board*`PostHook` function is not typically required. If you do not need this function, set `board.PostHookFunction` to `'NULL'`.

matlabroot`\toolbox\rtw\targets\xpc\target\build\xpcblocks\thirdpartydrivers\`*your_company_name_board*`_int.m`

See Also

`xpcPCIDevice`

PreHook

Run just before the interrupt service routine

Syntax

```
int __cdecl  your_company_name_boardPreHook(xpcPCIDevice *pciInfo);
```

Argument

pciInfo Pointer to the PciDevice structure.

Description

*your_company_name_board*PreHook runs just before the model-level interrupt service routine (either a function-call subsystem or entire model) is called.

Return

This function must check the status register on the board to determine if the board caused the interrupt. It returns one of the following:

- XPC_RUN_ISR — If the function determines that the board did cause the interrupt, the function must perform the required operation to stop the board from generating the interrupt. The function then returns this value.
- XPC_DROP_ISR — If the function determines that the board did not cause the interrupt, this function returns this value.

See Also

xpcPCIDevice

Start

Run as the last item in `mdlStart`

Syntax

```
void __cdecl your_company_name_boardStart(xpcPCIDevice *pciInfo);
```

Argument

`pciInfo` Pointer to the `PciDevice` structure.

Description

`your_company_name_board``Start` runs as the last item after all of the `mdlStart` functions. It is typically used to turn on interrupt generation.

See Also

`xpcPCIDevice`

Stop

Run at the beginning of `mdlTerminate`

Syntax

```
void __cdecl your_company_name_boardStop(xpcPCIDevice *pciInfo);
```

Argument

`pciInfo` Pointer to the `PciDevice` structure.

Description

`your_company_name_board``Stop` runs before the `mdlTerminate` function of the blocks in the model. It is typically used to turn off interrupt generation.

See Also

`xpcPCIDevice`

6

Custom Simulink Real-Time Driver Notes

- "S-Function Guidelines" on page 6-2
- "mdlStart and mdlTerminate Considerations" on page 6-4
- "DMA Considerations" on page 6-5
- "Passing Parameters" on page 6-6
- "Accessing Registers" on page 6-7

S-Function Guidelines

You implement Simulink Real-Time device driver blocks on the development computer using Simulink S-functions. An S-function is a set of subroutines that implements a function. You can write a Simulink Real-Time S-function in C, C++, or Fortran. You cannot write a Simulink Real-Time S-function in MATLAB code.

Simulink S-functions have a number of callback methods. For Simulink Real-Time drivers, you typically need to write C code for the following callback methods:

Method	Description
`mdlInitializeSizes`	Initializes the S-function with the number of inputs, outputs, states, parameters, and other characteristics.
`mdlInitializeSampleTimes`	Initializes the sample rates of the S-function.
`mdlStart`	Initializes the state vectors of this S-function and initializes the associated I/O module.
`mdlOutputs`	Computes the signals that this block emits.
`mdlTerminate`	Performs the actions required at termination of the simulation.

After you create the S-function, create a mask for it. See “Driver Mask Guidelines” on page 4-3.).

Of particular note when writing S-functions:

- Keep track of the input parameters the driver will require. When you create a mask for the driver, you will need to know this.
- Work vectors are not shared between runs. The S-function work variables are cleared after calling `mdlTerminate`. This implies that each time the S-function calls `mdlStart`, you must reinitialize the work variables.
- Declare the memory-mapped registers as `volatile`.
- An S-function is compiled into a MEX-file to run as part of the simulated model on the development computer. During code generation, the S-function calls the `mdlInitializeSizes` and `mdlInitializeSampleTimes` functions to determine the data structures that are used on the target. The same C file is also compiled with your real-time application to run on the target computer. Because of the following reasons, you must conditionally compile code for the development computer and for the target computer.

- The development computer runs Windows and the target computer runs the Simulink Real-Time kernel.
- The development computer does not have the same I/O modules as the target computer.

The preprocessor symbol `MATLAB_MEX_FILE` is defined when you compile for simulation (via `mex`). Undefine this symbol when compiling for the Simulink Real-Time environment. Use this symbol to conditionally compile development computer or target computer specific code. For example:

```
#ifdef MATLAB_MEX_FILE /* development computer */
 /* simulation code, typically nothing */
#else /* target computer */
 /* code to access I/O board */
# endif
```

If you want the code to run on both the development and target computers, do not conditionalize the code.

- Include the `xpctarget.h` file in your S-function.

  This provides definitions for the functions exported by the Simulink Real-Time kernel. The Simulink Real-Time kernel exports a number of functions for use in device drivers.

See “mdlStart and mdlTerminate Considerations” on page 6-4 for notes on specific uses for the callback methods.

mdlStart and mdlTerminate Considerations

When you load a real-time application onto a target computer, the driver executes the `mdlStart` callback method. If `mdlStart` completes and the software does not detect an error, the driver then executes `mdlTerminate`.

If `mdlStart` does not complete or the software detects an error, the real-time application does not execute `mdlTerminate`. (Typically, `mdlStart` might not complete if the application cannot find a referenced I/O board or if the board does not initialize.)

When the real-time application does start, it executes `mdlStart` again, then repeatedly executes `mdlOutputs`. At the end of execution, the application calls the `mdlTerminate` function.

With the above considerations, write `mdlStart` and `mdlTerminate` so that they cancel each other out. The `mdlTerminate` function should deallocate the resources allocated in `mdlStart`. For example, if you set an output to high in `mdlStart`, reset it to the default level in `mdlTerminate`. (Failure to reset the output causes a high output before the real-time application starts.) As another example, if, in the `mdlStart` function, you allocate memory, have `mdlTerminate` free the memory.

Although this description distinguishes between the driver initialization and real-time application start phases, you do not need to actually differentiate between them. If you do need to do so, use the `xpcIsModelInit` function. This function returns `1` while the model is initializing, and `0` otherwise.

DMA Considerations

If your board directly accesses system RAM, such as a DMA controller, you must allocate that memory using the `xpcAllocPhysicalMemory` function. This function allocates the buffer such that the buffer virtual address is the same as its physical address.

Passing Parameters

See “Passing Parameters to S-Functions”.

Accessing Registers

In this section...
"I/O Space" on page 6-7
"Memory-Mapped Space" on page 6-7

I/O Space

For registers in I/O space, use the Simulink Real-Time I/O read and write functions:

- Data read functions

```
uint32_T xpcInpDW(uint16_T port);  // read a 32 bit word
uint16_T xpcInpW(uint16_T port);   // read a 16 bit word
uint8_T  xpcInpB(uint16_T port);   // read an 8 bit byte
```

- Write functions

```
void xpcOutpDW(uint16_T port, uint32_T value);  // write 32 bits
void xpcOutpW(uint16_T port, uint16_T value);   // write 16 bits
void xpcOutpB(uint8_T port,  uint8_T value);    // write a byte
```

The port is the address value returned in the `BaseAddress` array.

To read a data value from a register, send the I/O space address of the data through the output (address) port using an `xpcOutp*` function, and then read the data from the input (data) port using an `xpcInp*` function.

To write a data value into a register, send the I/O space address of the data through the output (address) port using an `xpcOutp*` function, and then write the data through the input (data) port using an `xpcOutp*` function.

Memory-Mapped Space

For registers in memory-mapped space, dereference them through a pointer that contains the virtual address returned by the `xpcReserveMemoryRegion` function. Because modern compilers have aggressive optimizers, you must declare the pointer `volatile` so the compiler does not optimize out reads and writes using that pointer. The following pseudocode illustrates this using two methods: structure and array.

- Structure

```
struct bdregs {
    volatile int reg1;
    volatile int reg2;
    etc.
};

    struct bdregs *regs = pciInfo.VirtualAddress[1];

    regs->reg1 = 0x1234;  // Sets reg1 to that value
    regs->reg2 = 0x56789abc;
    etc.
```

If your I/O module uses registers with different lengths, it might be easier to use the structure method.

- Array

```
#define REG1   0
#define REG2   1
etc.

    volatile int *aregs = pciInfo.VirtualAddress[1];

    aregs[REG1] = 0x1234;
    aregs[REG2] = 0x56789abc;
```

7

Using the Simulink Real-Time Driver Authoring Tool

- "Driver Authoring Tool Basics" on page 7-2
- "Generating Custom Driver Templates" on page 7-3

Driver Authoring Tool Basics

Simulink Real-Time Driver Authoring Tool helps you create templates for simple custom device drivers. A simple custom device driver is one that does not perform DMA or interrupt processing. The Simulink Real-Time Driver Authoring Tool is not useful for these more complicated implementations.

Based on the inputs you provide to Simulink Real-Time Driver Authoring Tool, it can create a number of files, including the following. Of these files, you need to edit only the source C code file. You can also optionally edit the block mask file.

File	Description
driver_name.c	Template for the source C code for driver. You need to enter your C code.
driver_name.h	Header file for driver.
sfcn_driver_name.c	S-function file for driver. This file contains S-function callback methods and options for the driver.
sfcn_driver_name.tlc	Optional. Simulink Coder TLC code generation file. You typically need a `.tlc` file only if you want to inline your custom driver. See “Inlining Simulink Real-Time Drivers” on page 1-10 for further information. The Simulink Real-Time Driver Authoring Tool creates this file for you whether or not you want to inline the driver.
driver_block_name	Optional. Block mask model file for driver. After the Simulink Real-Time Driver Authoring Tool creates the supporting files, it creates the block mask for the driver and displays it in the Simulink model window. The tool creates this file only if you select the **MEX C file** check box.
sfcn_driver_name.mexw32	Optional. If you requested the creation of a C MEX file, the tool generates one for you.

Note: The Simulink Real-Time Driver Authoring Tool creates custom driver templates using the Legacy Code Tool (LCT). You do not need prior knowledge of the Legacy Code Tool to use the Simulink Real-Time Driver Authoring Tool. If you want to read about the Legacy Code Tool, see “Integrate C Functions Using Legacy Code Tool”.

Generating Custom Driver Templates

In this section...
"Using the Driver Authoring Tool" on page 7-3
"Setting Up Driver Variables" on page 7-3
"Saving the Configuration" on page 7-6
"Reloading the Configuration" on page 7-6
"Creating the C File Template" on page 7-6
"Creating a C MEX File for the Driver" on page 7-7
"Customizing the Device Driver Mask" on page 7-7

Using the Driver Authoring Tool

The prerequisites for creating a custom Simulink Real-Time device driver using the Simulink Real-Time Driver Authoring Tool are the same as those for creating a device driver manually. See "Expected Background" on page 1-3 and "Before You Start" on page 1-8 for further information.

The following sections assume that you have identified the following component specifications for the driver. See "Before You Start" on page 1-8 for guidelines for the following driver components, including their data type and size:
Input ports
Output ports
Parameters
Work variables

Setting Up Driver Variables

1. In the MATLAB Command Window, change folder to the one in which you want to save the driver code.
2. Start Simulink Real-Time Driver Authoring Tool. Type

   ```
   slrtdrivertool
   ```

   The Simulink Real-Time Driver Authoring Tool is displayed.
3. In the Main tab, enter:

- Driver name — The name for your driver. The tool will create supporting files using this string as the prefix. For example, type `testdriver`.
- Sample time — Select one of the following:
  - `Mask parameter` — If you want the block sample time to be settable as a block dialog box parameter (**Sample Time**).
  - `Inherited` — If you want the block to inherit its sample time from a connected block. No **Sample Time** parameter is displayed in the block dialog box.

4 If you have input ports for the block, click the **Input Ports** tab.

The **Inport Ports** tab is displayed.

5 Click the **Add** button. Enter your input port information in the following fields. Repeat for the rest of your input ports.

- **Variable** — Enter the name of the input. For example, `speed`.
- **Size** — Enter the maximum size number of storage locations to be allocated for the parameter. If you want this number to be a variable one, enter a value of 0. This setting means that you can pass an additional function argument that contains the size into the start, output, and/or terminate functions along with the port/parameter variable.
- **Type** — From the list, select the data type for the input port.
- **Output** — Always selected. Passes the input port value into the S-function `mdlOutputs` callback method.

6 If you have output ports for the block, click the **Output Ports** tab.

The **Output Ports** tab is displayed.

7 Click the **Add** button. Enter your output port information in the following fields. Repeat for the rest of your output ports.

- **Variable** — Enter the name of the output. For example, `speed`.
- **Size** — Enter the maximum size number of storage locations to be allocated for the size.
- **Type** — From the list, select the data type for the output port.
- **Output** — Always selected. Passes the output port value into the S-function `mdlOutputs` callback method.

8 If you have parameters for the block, click the **Parameters** tab.

The **Parameters** tab is displayed.

9 Click the **Add** button. Enter your parameter information in the following fields. Repeat for the rest of your parameters.

- **Variable** — Enter the name of the parameter. For example, `speed`.
- **Type** — From the list, select the data type for the parameter.
- **Size** — Enter the maximum size number of storage locations to be allocated for the parameter. If you want this number to be a variable one, enter a value of 0. This means that you can pass an additional function argument that contains the size into the start, output and/or terminate functions along with the port/parameter variable.
- **Start** — Select the check box if you want the parameter value to be passed into the S-function `mdlStart` callback method.
- **Output** — Select the check box if you want the parameter value to be passed into the S-function `mdlOutputs` callback method.
- **Terminate** — Select the check box if you want the parameter value to be passed into the S-function `mdlTerminate` callback method.

10 If you have work variables to be shared between the start, output, and terminate routines for the block, click the **Work Variables** tab.

The **Work Variables** tab is displayed.

11 Click the **Add** button. Enter your work variables information in the following fields. Repeat for the rest of your parameters.

- **Variable** — Enter the name of the work variable. For example, `speed`.
- **Type** — From the list, select the data type for the work variable.
- **Size** — Enter the maximum size of the work variable.
- **Start** — Select the check box if you want the work variable value to be passed into the S-function `mdlStart` callback method.
- **Output** — Select the check box if you want the work variable value to be passed into the S-function `mdlOutputs` callback method.
- **Terminate** — Select the check box if you want the work variable value to be passed into the S-function `mdlTerminate` callback method.

Saving the Configuration

The Simulink Real-Time Driver Authoring Tool allows you to save your configuration session as a MAT-file.

1. In the Simulink Real-Time Driver Authoring Tool, click the **Main** tab.
2. Click **Save settings**.

   The tool saves the `testdriver.mat` file in the current working folder.

You can iteratively change the configuration and resave the MAT-file as often as you like.

Reloading the Configuration

The Simulink Real-Time Driver Authoring Tool allows you to reload your configuration session as a MAT-file.

1. In the Simulink Real-Time Driver Authoring Tool, click the **Main** tab.
2. Click **Load settings**.

   The tool loads the `testdriver.mat` file into the tool.

Creating the C File Template

To generate a template for the driver C source code file:

1. In the Simulink Real-Time Driver Authoring Tool, click the **Main** tab.
2. Select **Generate C file template**.
3. Click the **Build** button.

   The tool creates the following files:

   - `testdriver.c`
   - `testdriver.h`
   - `sfcn_testdriver.c`
   - `sfcn_testdriver.tlc`
4. With your favorite editor, open the `testdriver.c` file and edit it. This is the source C code for your driver. The S-function code in `sfcn_testdriver.c` will reference this C file.

Creating a C MEX File for the Driver

To create a C MEX file for the driver, you can use either the Simulink Real-Time Driver Authoring Tool or the `mex` function.

Note: Use the Simulink Real-Time Driver Authoring Tool to build the C Mex file if you have not edited the C source code file (`testdriver.c`). If you have edited this file and want to keep those changes, do not use the Simulink Real-Time Driver Authoring Tool to build the driver. Doing so overwrites your changes to the C source code. Instead, use the `mex` function (see "Creating a C MEX File Using the mex Function" on page 7-7).

Creating a C MEX File Using the Authoring Tool

1. In the Simulink Real-Time Driver Authoring Tool, click the **Main** tab.
2. Select **Generate block and mask**.
3. Click the **Build** button.

   The tool creates the file `sfcn_testdriver.mexw32`.

Creating a C MEX File Using the mex Function

1. In the MATLAB Command Window, change folder to the one that contains the driver files.
2. Compile and link the MEX-file. For example:

   ```
   mex sfcn_testdriver.c testdriver.c
   ```

   This function creates the `sfcn_testdriver.win32mex` file.

Customizing the Device Driver Mask

The Simulink Real-Time Driver Authoring Tool creates a mask for the device driver. For guidelines on customizing this mask, see "Driver Mask Guidelines" on page 4-3. If you customize the mask, do not use the Simulink Real-Time Driver Authoring Tool again to build your files. Doing so overwrites the driver files and you lose your mask customizations.

I/O Structures — Alphabetical List

xpcPCIDevice

Type definition for PCI configuration space structure

Prototype

```
typedef struct xpcPCIDeviceStruct{
    uint32_T BaseAddress[6];
    uint32_T VirtAddress[6];
    uint32_T Length[6];
    uint16_T AddressSpaceIndicator[6];
    uint16_T MemoryType[6];
    uint16_T Prefetchable[6];
    uint16_T InterruptLine;
    uint16_T VendorId;
    uint16_T DeviceId;
    uint16_T SubDeviceId;
    uint16_T SubVendorId;
} xpcPCIDevice;
```

Header File

```
xpctarget.h
```

Members

`BaseAddress`	Physical base addresses that are assigned by the PCI BIOS.
`VirtAddress`	Virtual address of device. You can enter the return value from `xpcReserveMemoryRegion`. See "Description" on page 8-3 for details.
`Length`	Length of each region. This value contains the number of bytes that the board segment responds to during the configuration space read. This value might be larger than the space required by the registers as specified in the manufacturer documentation.

`AddressSpaceIndicator`	Indicates whether the board is I/O port mapped or memory-mapped. Values are one of the following. Verify this value in the manufacturer documentation. `0` Memory-mapped `1` I/O port mapped
`MemoryType`	Type of memory. This field is relevant only if `AddressSpaceIndicator` has a value of `0`. 0 Located anywhere in the 32-bit address space 1 Located below 1 MB 2 Located anywhere in the 64-bit address space
`Prefetchable`	Indicates whether or not the memory is prefetchable. Typically, this field is not required.
`InterruptLine`	Contains the assigned interrupt line, values between 0 and 15. The BIOS assigns this value. You need this value only if you are writing an interrupt driver.
`VendorId`	Contains vendor ID.
`DeviceId`	Contains device ID.
`SubDeviceId`	Contains subdevice ID.
`SubVendorId`	Contains subvendor ID.

Description

The `xpcPCIDevice` structure defines the PCI configuration space structure. The following are additional notes on the `BaseAddress` field:

- The PCI specification allows the definition of up to six different base addresses (addressable regions). Most boards respond to one or two of these addresses. Base addresses are filled in during the BIOS plug and play initialization, before the Simulink Real-Time kernel starts to execute. The designer of the board decides how many address spaces are defined and what they are used for. Many boards use one address space to contain all of the registers for the board, other boards separate functions into different address spaces. See the board manufacturer documentation for this information.

- For memory-mapped segments, call the `xpcReserveMemoryRegion` function to convert the physical address in `BaseAddress` to a virtual address that is suitable for the CPU to read and write the segment. You can then optionally save this address in the `VirtAddress` field. You might want to save the address if you have several segments and you want to pass them to a board access library.

See Also

`xpcGetPCIDeviceInfo`, `xpcShowPCIDeviceInfo`

xpcTime

Type definition of time structure

Prototype

```
typedef struct xpcTime64Struct{
    uint32_T NanoSecondsLo;
    uint32_T NanoSecondsHi;
} xpcTime64;

typedef union xpcTimeStruct{
    xpcTime64 U64;
    //uint64_T NanoSeconds;
} xpcTime;
```

Header File

`xpctarget.h`

Members

`U64.NanoSecondsLo`	Bottom 32 bits of 64-bit value.
`U64.NanoSecondsHi`	Top 32 bits of 64-bit value.

Description

The `xPCTime` structure holds the time value in nanoseconds, as a 64-bit integer. *`NanoSecondsLo`* and *`NanoSecondsHi`* hold the lower and upper 32 bits, respectively. The `xpcGetElapsedTime` and `xpcSubtractTime` functions use this structure to return time values.

See Also

`xpcGetElapsedTime`, `xpcSubtractTime`

I/O Functions — Alphabetical List

xpcAllocPhysicalMemory

Allocate physical memory

Prototype

```
void *xpcAllocPhysicalMemory(uint32_T numBytes)
```

Header File

```
xpctarget.h
```

Arguments

numBytes	Allocate specified number of bytes of memory.

Description

The `xpcAllocPhysicalMemory` function allocates the requested bytes of physical memory. Functions such as `malloc` only return virtual memory.

`xPCAllocPhysicalMemory` allocates physical memory, where physical memory is the same as the virtual address. Use this function only for allocations requiring direct access to physical memory, such as allocations for DMA transfers.

See Also

`xpcFreePhysicalMemory`

xpcBusyWait

Wait for specified length of time in seconds

Prototype

```
void xpcBusyWait(real_T seconds)
```

Header File

```
xpctarget.h
```

Arguments

seconds	Length of time to wait, in seconds.

Description

The `xpcBusyWait` function waits for the specified number of seconds. This function blocks this specified amount of time.

xpcFreePhysicalMemory

Free physical memory

Prototype

```
void xpcFreePhysicalMemory(const void *physical)
```

Header File

```
xpctarget.h
```

Arguments

physical	Free specified memory.

Description

The `xpcFreePhysicalMemory` function frees the specified section of physical memory.

See Also

`xpcAllocPhysicalMemory`

xpcGetElapsedTime

Return time since system boot

Prototype

```
real_T xpcGetElapsedTime(xpcTime *upTime)
```

Arguments

upTime	Pointer to an `xpcTime` structure.

Description

The `xpcGetElapsedTime` function returns the time since the system was last booted, in seconds. You can get this time in nanoseconds by passing a pointer to a previously allocated `xpcTime` structure. If you do not want the time in nanoseconds, you can pass a NULL pointer for the `upTime` argument.

See Also

`xpcTime`, `xpcSubtractTime`

xpcGetPCIDeviceInfo

Return information for PCI device

Prototype

```
int32_T xpcGetPCIDeviceInfo (uint16_T vendorId, uint16_T deviceId, uint16_T subVendorId
```

Arguments

vendorId	Enter the vendor ID.
deviceId	Enter the device ID.
subVendorId	Enter the subvendor ID.
subDeviceId	Enter the subdevice ID.
bus	Enter the device bus.
slot	Enter the slot that contains the device.
pciInfo	Pointer to the `PciDevice` structure.

Header File

`xpctarget.h`

Description

The `xpcGetPCIDeviceInfo` function fills the structure, *`pciInfo`*, with the PCI configuration information for the specified PCI device. This information includes base address, registers, IRQ, and so forth, from the PCI BIOS. It uses the vendor and device IDs and, optionally, the subvendor and subdevice IDs to search for the board.

If you specify `XPC_NO_SUB` for subvendor or subdevice ID, or `XPC_NO_BUS_SLOT` for device slot, `xpcGetPCIDeviceInfo` returns the first matching board that it finds in the PCI BIOS. If you specify a bus and a slot value, the function returns only a board with matching IDs found at that bus or slot.

Supply valid vendor and device IDs. If you specify values other than `XPC_NO_SUB` for subvendor and subdevice IDs, the function matches the board using all four ID parameters. To find a board using only vendor ID and device ID, use `XPC_NO_SUB` for *`subDeviceId`* and `XPC_NO_SUB` for *`subVendorId`*.

This function returns `0` if it executes without detecting an error. Otherwise, it returns a nonzero value.

See Also

`xpcPCIDevice` `xpcShowPCIDeviceInfo`

xpcInpB

Read 8-bit data from data register

Syntax

```
byte_data = xpcinpB(data_port_id)
```

Description

`byte_data = xpcinpB(data_port_id)` returns a byte (8-bit) value from I/O data port `data_port_id`. To read a register:

1. Send the I/O space address of the data through the output (address) port using `xpcOutpB`.
2. Read the data through the input (data) port using `xpcInpB`.

Header file: `xpctarget.h`

Examples

Read current day of month

The I/O port address of the real-time clock (RTC) is `0x70` for output and `0x71` for input. The RTC day-of-month field is at `0x07`. Set the device register address for the day-of-month field, and then read the day-of-month value.

Initialize variables for address port, data port, and day-of-month field.

```
uint16_T address_port_id = 0x70;
uint16_T data_port_id = 0x71;
uint8_T day_of_month_addr = 0x07;
```

Set device register address of day-of-month field.

```
xpcOutpB(address_port_id, day_of_month_addr);
```

Read day of month value.

```
uint8_T day_of_month = xpcInpB(data_port_id);
```

Input Arguments

`data_port_id` — I/O port bus address of device

uint16_T

I/O port bus address of device to read data from memory.

Example: 0x71

Output Arguments

`byte_data` — Value returned from device register

uint8_T

The register returns a byte (8-bit) value.

See Also

`xpcOutpB`

Introduced before R2006a

xpcInpDW

Read 32-bit data from data register

Syntax

```
dword_data = xpcinpDW(data_port_id)
```

Description

`dword_data = xpcinpDW(data_port_id)` returns a double word (32-bit) value from I/O data port `data_port_id`. To read a register:

1. Send the I/O space address of the data through the output (address) port using `xpcOutpDW`.
2. Read the data through the input (data) port using `xpcInpDW`.

Header file: `xpctarget.h`

Examples

Read current second and minute into 32-bit word

The I/O port address of the real-time clock (RTC) is `0x70` for output and `0x71` for input. The RTC current-second and current-minute fields are at addresses `0x00` and `0x02`. Set the device register address to point to the current-second field, and then read one 32-bit word. The device returns the second and minute as the first and third bytes.

Initialize variables for address port, data port, and current-second field.

```
uint16_T address_port_id = 0x70;
uint16_T data_port_id = 0x71;
uint32_T current_second_addr = 0x00;
```

Set register address of current second field.

```
xpcOutpDW(address_port_id, current_second_addr);
```

Read current second and current minute.

```
uint32_T second_and_minute = xpcInpDW(data_port_id);
uint8_T second = ((uint8_T*)(&second_and_minute))[0];
uint8_T minute = ((uint8_T*)(&second_and_minute))[2];
```

Input Arguments

`data_port_id` — I/O port bus address of device

uint16_T

I/O port bus address of device to read data from memory.

Example: 0x71

Output Arguments

`dword_data` — Value returned from device register

uint32_T

The register returns a double word (32-bit) value.

See Also

`xpcOutpDW`

Introduced before R2006a

xpcInpW

Read 16-bit data from data register

Syntax

```
word_data = xpcinpW(data_port_id)
```

Description

`word_data = xpcinpW(data_port_id)` returns a word (16-bit) value from I/O data port `data_port_id`. To read a register:

1. Send the I/O space address of the data through the output (address) port using `xpcOutpW`.
2. Read the data through the input (data) port using `xpcInpW`.

Header file: `xpctarget.h`

Examples

Read current month and year into 16-bit word

The I/O port address of the real-time clock (RTC) is `0x70` for output and `0x71` for input. The RTC current month and current year fields are at `0x08` and `0x09`. Set the device register address to point to the current month value, and then read one 16-bit word. The device returns the month and year as the first and second bytes.

Initialize variables for address port, data port, and current-month field.

```
uint16_T address_port_id = 0x70;
uint16_T data_port_id = 0x71;
uint16_T current_month_addr = 0x08;
```

Set register address of current-month field.

```
xpcOutpW(address_port_id, current_month_addr);
```

Read month and year.

```
uint16_T month_and_year = xpcInpW(data_port_id);
uint8_T month = ((uint8_T*)(&month_and_year))[0];
uint8_T year = ((uint8_T*)(&month_and_year))[1];
```

Input Arguments

`data_port_id` — I/O port bus address of device

uint16_T

I/O port bus address of device to read data from memory.

Example: 0x71

Output Arguments

`word_data` — Value returned from device register

uint16_T

The register returns a word (16-bit) value.

See Also

`xpcOutpW`

Introduced before R2006a

xpcIsModelInit

Return real-time application load state

Prototype

```
boolean_T xpcIsModelInit(void)
```

Header File

```
xpctarget.h
```

Arguments

none

Description

The `xpcIsModelInit` function returns a Boolean value to indicate the real-time application load state:

- `true` — While real-time application is loading
- `false` — Start of real-time application execution

You can call this function from the `mdlStart` and `mdlTerminate` callbacks.

See Also

“mdlStart and mdlTerminate Considerations” on page 6-4

xpcOutpB

Write 8-bit data to data register

Syntax

```
xpcOutpB(data_port_id, byte_data)
```

Description

`xpcOutpB(data_port_id, byte_data)` writes a byte (8-bit) value into data port `data_port_id`. To write a register:

1. Send the I/O space address of the data register through the output (address) port using `xpcOutpB`.
2. Write the data through the input (data) port using `xpcOutpB`.

Header file: `xpctarget.h`

Examples

Change current day of month

The I/O port address of the real-time clock (RTC) is `0x70` for output and `0x71` for input. The RTC day-of-month field is at `0x07`. Set the device register address for the day-of-month field, and then write the new value.

Initialize variables for address port, data port, and day-of-month field.

```
uint16_T address_port_id = 0x70;
uint16_T data_port_id = 0x71;
uint8_T day_of_month_addr = 0x07;
```

Set register address of day-of-month field.

```
xpcOutpB(address_port_id, day_of_month_addr);
```

Change day of month.

```
uint8_T day_of_month = 29;
xpcOutpB(data_port_id, day_of_month);
```

Input Arguments

`data_port_id` — I/O port bus address of device

uint16_T

I/O port bus address of device to write data into memory.

Example: 0x70

`byte_data` — Value written into device register

uint8_T

Byte (8-bit) value written into the register.

Example: 28

See Also

`xpcInpB`

Introduced before R2006a

xpcOutpDW

Write 32-bit data to data register

Syntax

```
xpcOutpDW(data_port_id, double_word_data)
```

Description

`xpcOutpDW(data_port_id, double_word_data)` writes a double word (32-bit) value into data port `data_port_id`. To write a register:

1. Send the I/O space address of the data register through the output (address) port using `xpcOutpDW`.
2. Write the data through the input (data) port using `xpcOutpDW`.

Header file: `xpctarget.h`

Examples

Change current second and minute using 32-bit word

The I/O port address of the real-time clock (RTC) is `0x70` for output and `0x71` for input. The RTC current-second and current-minute fields are at addresses `0x00` and `0x02`. Set the device register address to point to the current-second field. Write one 32-bit word containing the new second and minute.

Initialize variables for address port, data port, and current-second field.

```
uint16_T address_port_id = 0x70;
uint16_T data_port_id = 0x71;
uint32_T current_second_addr = 0x00;
```

Set register address of current second field.

```
xpcOutpDW(address_port_id, current_second_addr);
```

Read current second and current minute, along with intervening fields.

```
uint32_T second_and_minute = xpcInpDW(data_port_id);
```

Change current second and current minute and send corrected field back.

```
((uint8_T *)(&second_and_minute))[0] = 1;
((uint8_T *)(&second_and_minute))[2] = 0;
xpcOutpDW(data_port_id, second_and_minute);
```

Input Arguments

`data_port_id` — I/O port bus address of device

uint16_T

I/O port bus address of device to write data into memory.

Example: 0x70

`double_word_data` — Value written into device register

uint32_T

Double word (32-bit) value written into the register.

Example: `0xFFFFFFFF`

See Also

`xpcInpDW`

Introduced before R2006a

xpcOutpW

Write 16-bit data to data register

Syntax

```
xpcOutpW(data_port_id, word_data)
```

Description

`xpcOutpW(data_port_id, word_data)` writes a word (16-bit) value into data port `data_port_id`. To write a register:

1. Send the I/O space address of the data register through the output (address) port using `xpcOutpW`.
2. Write the data through the input (data) port using `xpcOutpW`.

Header file: `xpctarget.h`

Examples

Change current month and year using 16-bit word

The I/O port address of the real-time clock (RTC) is `0x70` for output and `0x71` for input. The RTC current month and current year fields are at `0x08` and `0x09`. Set the device register address to point to the current month value. Write one 16-bit word containing the new month and year.

Initialize variables for address port, data port, and current-month field.

```
uint16_T address_port_id = 0x70;
uint16_T data_port_id = 0x71;
uint16_T current_month_addr = 0x08;
```

Set register address of current-month field.

```
xpcOutpW(address_port_id, current_month_addr);
```

Change month and year.

```
((uint8_T *)(&new_month_year))[0] = 2;
((uint8_T *)(&new_month_year)[1] = 29;
xpcOutpW(data_port_id, new_month_year);
```

Input Arguments

`data_port_id` — I/O port bus address of device

uint16_T

I/O port bus address of device to write data into memory.

Example: 0x70

`word_data` — Value written into device register

uint16_T

Word (16-bit) value written into the register.

Example: `0x128`

See Also

`xpcInpW`

Introduced before R2006a

xpcReserveMemoryRegion

Return virtual address that corresponds to physical address and mark region as readable/writable

Prototype

```
void * xpcReserveMemoryRegion(const void *physical, uint32_T
numBytes, uint32_T access)
```

Arguments

physical	Starting address of the memory region to be reserved. Typically, you obtain this address from one of the PCI base address registers.
numBytes	Size of region to be located, in bytes.
access	Type of access, limited to `XPC_RT_PG_USERREADWRITE` (read/write).

Return

The `xpcReserveMemoryRegion` function returns the virtual address to use to access the physical address.

Description

This function reserves a region of physical memory (as returned by the PCI BIOS) and returns the corresponding virtual address. You can later use the virtual address for pointer addressing.

You can call this function multiple times with the same address. A call to this function with an already reserved area returns the same virtual address.

The required size differs from board to board. You can obtain the required number of bytes from the register programming manual of the particular board. This size is typically a multiple of a page (4096 bytes).

xpcShowPCIDeviceInfo

Display contents of `PCIDevice` structure

Prototype

```
void xpcShowPCIDeviceInfo(xpcPCIDevice *pciInfo)
```

Arguments

`pciInfo`	Pointer to the `xpcPCIDevice` structure.

Description

This debugging function displays the contents of the `PCIDevice` structure pointed to by *`pciInfo`*. You can use this function with the `xpcGetPCIDeviceInfo` function to display the contents of the `xpcPCIDevice` structure.

Note: Remove this function from the driver before using the driver to interact with an I/O module.

See Also

`xpcGetPCIDeviceInfo`

xpcSubtractTime

Return difference between two times

Prototype

```
real_T xpcSubtractTime(xpcTime *time,
const xpcTime *time2, const xpcTime *time1)
```

Arguments

time	Pointer to an xpcTime structure.
time2	Enter the time to subtract.
time1	Enter the time to subtract from.

Description

xpcSubtractTime returns the difference between *time1* and *time2* (*time2* - *time1*), in seconds. You can get this time in nanoseconds by passing a pointer to a previously allocated xpcTime structure. If you do not want the time in nanoseconds, pass a NULL pointer for the time argument.

See Also

xpcTime, xpcGetElapsedTime