xPC Target™ 4
Device Drivers

MATLAB®
&SIMULINK®

The MathWorks™
Accelerating the pace of engineering and science
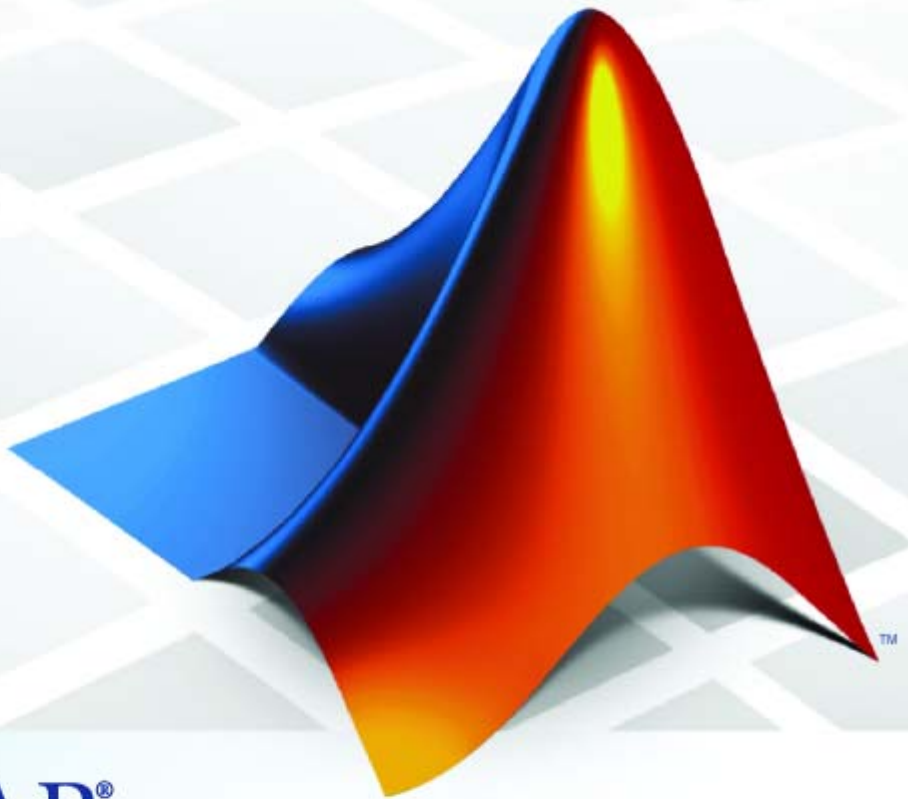
How to Contact The MathWorks

	www.mathworks.com	Web
	comp.soft-sys.matlab	Newsgroup
	www.mathworks.com/contact_TS.html	Technical Support
@	suggest@mathworks.com	Product enhancement suggestions
	bugs@mathworks.com	Bug reports
	doc@mathworks.com	Documentation error reports
	service@mathworks.com	Order status, license renewals, passcodes
	info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

xPC Target™ Device Drivers Guide

© COPYRIGHT 2007–2009 by 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

The 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)

Contents

Custom xPC Target Driver Notes

6

Creating Custom Drivers Using the xPC Target Driver Authoring Tool

7

8

I/O Structures — By Category

9

I/O Structures — Alphabetical List

10

I/O Functions — By Category

11

I/O Functions — Alphabetical List

1

Customizing xPC Target Drivers

- "Introduction" on page 1-2
- "Before You Start" on page 1-8
- "Creating a Custom Driver" on page 1-11
- "Debugging Notes" on page 1-16

Introduction

In this section...

xPC Target Drivers

The xPC Target™ software provides device drivers for a variety of third-party boards. xPC Target users access these drivers as Simulink® blocks from the xPC Target library (xpclib). If you have a board for which the xPC Target software does not supply a driver, you can write your own. This guide provides guidelines for writing custom xPC Target device drivers.

The xPC Target driver library contains drivers that support third-party boards with many I/O capabilities and applications. 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 xPC Target block library if:

- No xPC Target driver exists for your I/O needs.

- You are unable to use a board that the xPC Target software supports.

- You need to extend the functionality of an existing xPC Target driver.

- The MathWorks xPC Target team will not write a device driver for your board.

Restrictions on Customizing xPC Target Drivers

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

The xPC Target software installs its own kernel on the target PC. 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 xPC Target kernel cannot resolve. The xPC Target 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 The 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 xPC Target kernel.

Expected Background

This guide assumes that you are already knowledgeable about writing device drivers. It describes the steps specific to writing device drivers for the xPC Target environment. To write your own device drivers for the xPC Target 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 Real-Time Workshop

- Understanding of I/O hardware. Because of the real-time nature of the xPC Target software, you must develop drivers with minimal latency. Most drivers access the I/O hardware at the lowest possible level (register programming). Therefore, you must understand how to control the board with register information.

- Knowledge of port and memory I/O access over various buses. You need this information to access I/O hardware at the register level.

- Knowledge of PC hardware fundamentals and internals

Resources for Customizing xPC Target Drivers

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

References

The following MathWorks documentation provides information that you can refer to when customizing xPC Target drivers:

See...	For...
Simulink User's Guide	Overall description of the Simulink environment and how the Simulink software performs simulations.
MATLAB® External Interfaces	How to write MATLAB MEX-files.

See...	For...
Writing S-Functions	How to write MATLAB C-MEX S-functions (noninlined S-functions). Note the following references in this guide: • "S-Function Callback Methods — Alphabetical List" the Simulink software invokes these methods when simulating a model with S-functions. Real-Time Workshop uses the same methods in generated real-time applications. • "SimStruct Functions — Alphabetical List" Contains detailed descriptions of the functions that access the fields of an S-function's simulation data structure (`SimStruct`). S-function callback methods use these functions to store and retrieve information about an S-function.
Real-Time Workshop® Target Language Compiler	How to write target language compiler (TLC) files to inline S-function drivers. This is an optional reference and depends on whether or not you choose to inline your driver.
Real-Time Workshop User's Guide	Overall description of Real-Time Workshop 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 xPC Target device drivers as a reference for your custom drivers. Refer to the following directory:

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

Note In this directory, you might notice that some drivers use outdated xPC Target driver functions. For the current functions to use, see "xPC Target Exported Functions" on page 1-6.

xPC Target Exported Functions

The xPC Target 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 time-out intervals. Use only the functions documented in this guide. The guidelines in this document are not applicable if you are using an xPC Target software version prior to xPC Target software version 3.2 (R2007a). See Chapter 11, "I/O Functions — Alphabetical List", for a description of these functions.

Third-Party Directory

The xPC Target software provides the following directory to help you integrate your custom driver.

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

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

What Makes Up an xPC Target Driver?

An xPC Target device driver is an S-function with functions that access an I/O board.

Like any device driver, an xPC Target driver interfaces between the user and an I/O device. Unlike typical device drivers, xPC Target device drivers:

- Can have the following parts
  - Driver code, that is C code written as an S-function using exported xPC Target kernel functions (see "xPC Target Exported Functions" on page 1-6)
  - Optional Simulink block interface (Simulink mask) that users use to configure the device and access output
  - Optional M-file 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

- Are configured like any other Simulink block

Underlying driver code (C-file)

```
// addiamondmm32.c - xPC Target, non-inlined S-function driver
// for A/D section of Diamond Systems MM-32 boards
// Copyright 1996-2008 The MathWorks, Inc.

#define      S_FUNCTION_LEVEL      2
#undef       S_FUNCTION_NAME
#define      S_FUNCTION_NAME       addiamondmm32

#include     <stddef.h>
#include     <stdlib.h>

#include     "simstruc.h"

#ifdef       MATLAB_MEX_FILE
#include     "mex.h"
#endif

#ifndef      MATLAB_MEX_FILE
#include     <windows.h>
#include     "xpctarget.h"
#endif
```

MM-32
Diamond
Analog Input   1

Terminator
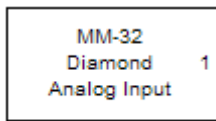
MM-32

Block mask
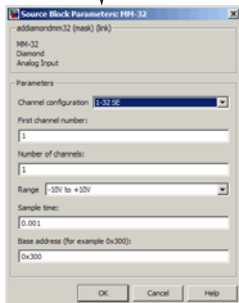
Mask editor
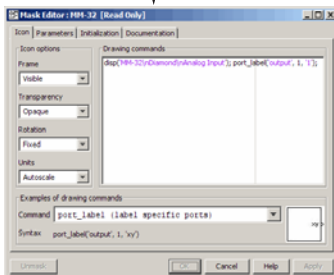
Mask initialization (M-file)

```
function [baseDec, maskDisplay, maskDescription] = ...
        maddiamondmm32(phase, configuration, firstChan, numChans, range,

% maddiamondmmx - Mask Initialization function for Diamond Systems MM-32
% Copyright 1996-2007 The MathWorks, Inc.
%    $Revision: 1.1.2.5 $  $Date: 2007/11/29 13:38:28 $

    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', 'LookUnderMask
        if length(blocks) > 1
            error('xPCTarget:DiamondMM32:Block',...
                  'Only one Diamond Systems MM-32 A/D block per physical
        end
        return
    end
```

Anatomy of an xPC Target™ 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 xPC Target 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

- "xPC Target Exported Functions" on page 1-6

- "Third-Party Directory" 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 Real-Time Workshop.

Driver Types

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

Bus Types and Register Access

The xPC Target software supports the two standard PC 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 xPC Target 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 any memory-mapped region.
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 xPC Target software only permits use of the memory address between 0xA0000 and 0xFFFFF
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 xPC Target Drivers

You can choose to inline or not inline xPC Target drivers. Note the distinction between Simulinkand Real-Time Workshop conditional compilation. Writing a device driver as an inlined S-function ensures that the driver can coexist with xPC Target device drivers.

Inlining drivers allows you to customize code generated from Real-Time Workshop. If you choose to create inlined drivers, you must use the Real-Time Workshop 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 how to create a custom device driver. For additional notes on writing custom xPC Target drivers, see Chapter 6, "Custom xPC Target Driver Notes". For a description of a tool that helps you create simple custom drivers, see Chapter 7, "Creating Custom Drivers Using the xPC Target Driver Authoring Tool". A simple custom device driver is one which performs no DMA or interrupt handling.

Note You might need administrative privileges to add a custom device driver to the xPC Targetsystem.

1 Write your driver in C, using the approved I/O functions as appropriate (see Chapter 9, "I/O Structures — Alphabetical List" and Chapter 11, "I/O Functions — Alphabetical List"). 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.

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

3 Create an M-file 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\diamondmm32.m

4 Open the Simulink Library Browser and create a new library, for example, *your_company_name*lib.mdl (see "Creating Block Libraries" in the "Working with Block Libraries" in the chapter of *Simulink User's Guide*). Use a unique library name to prevent conflicts with other libraries.

5 In the new library, create an S-function block. From the Simulink Library Browser, drag an S-Function block to the new library.

6 Configure the S-Function block.

   a In the new library, right-click the S-Function block and select **S-Function Parameters**.

   b In **S-function name**, enter the name, without extension, of the driver. For example, addiamondmm32. (This is the driver C-file you created in step 1).

   c In **S-function parameters**, enter the parameters you defined for the driver. The parameter names you enter here must match the names you will later enter for the driver block mask (through the **Parameters** and **Initialization** panes of the Mask Editor dialog box). For example, firstChan, numChans, range, sampleTime, baseDec. Step 7 describes the block mask creation.

   d Leave the **S-function modules** parameter with the default value, unless you need to separate your driver C-file into multiple files. If that is the case, see "Specifying Additional Source Files for an S-Function" in the "Writing S-Functions for Real-Time Workshop Code Generation" chapter of the *Real-Time Workshop User's Guide*.
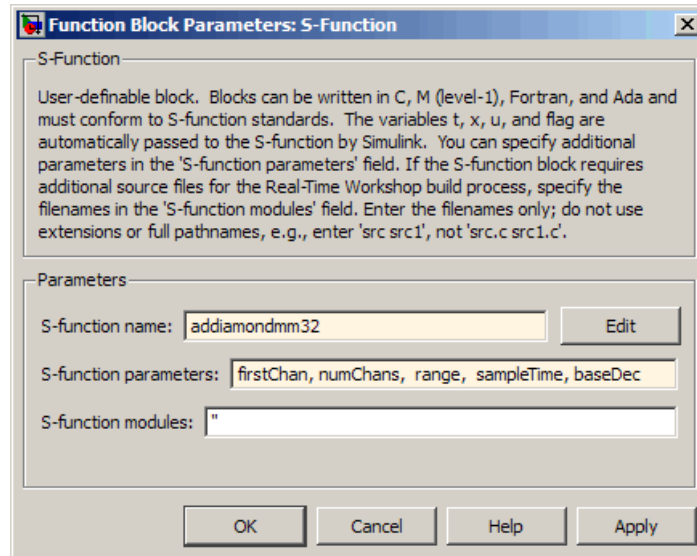
7 Double-click the S-Function block and create a block mask (see Chapter 4, "Masking Drivers").

8 Save and close the S-Function block.

9 At the bottom of the S-Function block, enter a block name. For example, `MM-32`.
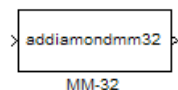
10 Save and close the library.

11 To make your new library visible in the Simulink Library Browser, move it to

> *matlabroot*\toolbox\rtw\targets\xpc\target\build\xpcblocks\thirdpartydrivers

12 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.

• Ensure that out.IsFlat is set to 0.

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

Note Ensure that you create a function that calls the out structure.

---

13 (Optional for PCI boards) To enable the getxpcpci function to account for 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, change the ID number as necessary. 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';
```

e Save and close the file.

f To confirm your entries, type `getxpcpci('all')` in the MATLAB Command Window.

14 Ensure that all your driver files, including include files, are in the directory:

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

Ensure that these files have unique names to prevent conflicts.

15 To update the directories that you added, at the MATLAB Command Window, type

```
rehash toolbox
```

When you are done, your library will appear in the Simulink Library Browser with `xPC Target:` added to the beginning of the library name.

```
xPC Target: your_company_namelib Blockset
```

Debugging Notes

While developing your custom driver, you can use `printf` statements in your code. This displays output on the left-hand side of the target PC monitor. If your `printf` statements scroll off the monitor, consider disabling the target PC scope area to provide more display area for these statements:

1 At the MATLAB Command Window, type `xpcexplr` to start xPC Target Explorer.

2 In xPC Target Explorer, navigate to *target_PC_node* > **Configuration > Appearance**.

3 Clear the **Enable target scope** check box.

4 Recreate the target boot disk and reboot the target PC.

The scope area on the target PC monitor no longer appears.

5 Continue with the device driver development.

2

PCI Drivers

- "PCI Bus Considerations" on page 2-2
- "Sample PCI Device Driver" on page 2-8

PCI Bus Considerations

Introduction

When writing xPC Target 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 PC BIOS automatically assigns a conflict-free set of resources to any PCI device 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 xPC Target software provides functions to accomplish this (see the "PCI Configuration Information" on page 10-2 functions).

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 any one 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 PC 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 xPC Target software provides functions that allow the driver to access this space.

• Vendor and device ID — The driver searches all 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 do not always uniquely identify a board. For example, all 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.

2-3

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 the parameters	Looks for a device that matches all the 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

xPC Target 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 there is only one board of
this type in the target */
/* If there are multiple boards of this type 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 necessary */


/* 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 pointer of the appropriate size to the memory. For example:

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

Note that you will want to use the `volatile` keyword here; otherwise, the compiler might optimize away accesses to this location.

2 Set its 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

For example PCI device driver code, see

```
matlabroot\toolbox\rtw\targets\xpc\target\build\xpcblocks\dikpc
i1800.c
```

This driver illustrates digital input driver code for the Keithley 1800 series
PCI devices.

Note Remember to enter the C-file name (without the extension) as the
S-function name for the S-Function block.

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 3-3

Introduction

When writingxPC Target 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 ThexPC Target kernel does not support ISA and PC/104 PNP boards. This means that you can write xPC Target 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 if there are conflicts.

• Memory mapped

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

The xPC Target 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.

4

Masking Drivers

Creating Driver Subsystem Masks

This chapter describes guidelines for creating a Simulink block user interface (mask) for the S-Function block associated your driver. A mask defines the menu items that will be passed to the S-function. The mask can call an M-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 an xPC Target device driver:

1 Create an optional M-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 necessary. 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 xPC Target driver. You should already know how to create masked blocks. See the "Working with Block Masks" chapter in *Simulink User's Guide* for further information.

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 appropriate descriptive names for these parameters.

- 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 readable and appropriate variable names.

- As necessary and appropriate, 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.

- Ensure that the title beneath the driver block terminates with a blank space. This is because if a model contains more than one block of any given type, Simulink appends a number to the title under the block. Adding a blank space to the end of the label ensures readability.

- Name the block appropriately so that it indicates the purpose of the driver.

- If you want to link help information to the mask **Help** button, see "Block Help Field" in the "Working with Block Masks" chapter of *Simulink User's Guide* for details.

- From within the mask, you can call a custom M-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 hardware. 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:

- The recommended way is to call the `find_system` function from the block `InitFcn` callback function. There are two phases of M-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 hardware. 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 xPC Target 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 M-file for it, be sure to:

- Check the text of each error message for spelling and appropriateness.
- (Optional) Use a coding standard indentation such as four or eight spaces with no tabs.
- Copy your new blocks into a custom directory 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 hardware I/O under as many conditions as possible.

Sample Driver Mask

The following is the block mask for the Diamond MM-32 Analog Input block.
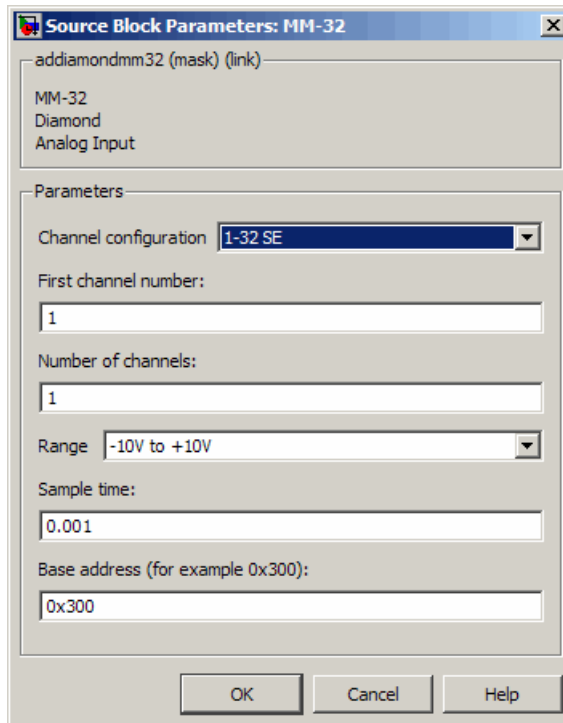
If you were to create this mask as a new mask, you would:

1 Open the Mask Editor for the block.

2 Select the **Parameters** tab and click the **Add** button on the left three times.

Three blank lines appear in the Dialog parameters section. Fill them in as follows, starting with the first line:

- In the **Prompt** column, enter

  Channel configuration

  First channel number:

```
Number of channels:

Range

Sample time:

Base address (for example 0x300):
```

- In the **Variable** column, enter the parameter names. Be sure that these names match the **S-function parameters** field of the S-Function block.

  ```
  configuration

  firstChan

  numChans

  range

  sampleTime

  base
  ```

- In the **Type** column, select:

  ```
  popup

  edit

  edit

  popup

  edit

  edit
  ```

- In the **Evaluate** and **Tunable** columns, ensure that the first five check boxes of **Evaluate** and all the check boxes for **Tunable** are selected.
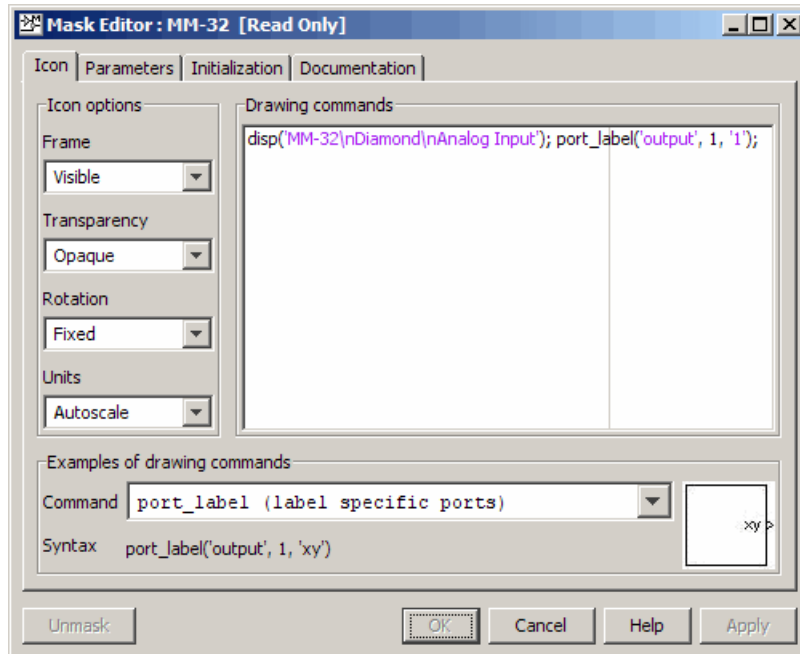
3 Select the **Initialization** tab. The tab displays the dialog variables you entered in the **Parameters** tab.

4 In the **Initialization** code section, type

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

where `maddiamondmm32` references the `maddiamondmm32.m` file for the driver.

In this M-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.

5 Select the **Documentation** tab. This tab contains three fields, **Mask type**, **Mask description**, and **Mask help**.

In the **Mask type** field, enter the type of driver. For example:

```
addiamondmm32
```

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

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

In the **Mask help** field, if you are providing any online documentation to associate with the **Help** button, you can call that online documentation from this field. See "Block Help Field" in the "Working with Block Masks" chapter of *Simulink User's Guide* for details.

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

After you create the block mask, you can 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 **Block Properties** from the drop-down list.

2 Select the **Callback** tab from the dialog. From the list, select `InitFcn`. Enter M-code in the edit box.

One convention is to use the same function that is used at mask initialization time (for example, `maddiamondmm32.m`), but with a single argument that indicates that this is being called at `InitFcn` time.

```
maddiamondmm32(1)
```

At `InitFcn` time, no variables in the mask exist yet. You cannot pass them into the `InitFcn`. However, in the function, you can get the string values using the `get_param` function. 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('xPCTarget: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
```

5

Interrupt Support

xPC Target Interrupts

In this section...
"Introduction" on page 5-2
"Interrupt Processing in the xPC Target Environment" on page 5-2

Introduction

If your device supports interrupts, you can use the procedures in this chapter to add your custom interrupt functions to the xPC Target framework.

Your users can use interrupts in xPC Target applications in one of the following ways:

- Use the interrupt with the xPC Target 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 **Real-Time Workshop > xPC Target options** pane.

Note Although users can use interrupts in one of two ways, you program for these interrupts using the same procedure, as described in "Adding Interrupt Support" on page 5-7. However, before you start programming the interrupts, see "Interrupt Processing in the xPC Target Environment" on page 5-2 for a description of the flow of xPC Target interrupt processing.

Interrupt Processing in the xPC Target Environment

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

1 Call all mdlStart routines in block execution order.

2 Call the Start function, if one exists.

3 Allow background graphics and network tasks to run until an interrupt occurs.

The following illustrates the flow of processing once a hardware interrupt occurs. This is background information to help you understand the context in which the interrupt functions run.

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 hardware 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 on page 5-5.

- If this board is configured to run the model, and this board did interrupt, send a wakeup call to the model thread. The model thread does not immediately execute. It waits until after all 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.

If no other device exists, generated code goes to step 6.

6 Returns from the interrupt.

The xPC Target kernel now runs the highest priority thread. The highest priority ready thread is the model if a wakeup call was sent to it.

Note The **Allow preemption of function call subsystem** check box has no effect. Interrupts are never enabled when the function-call subsystem is executed.

Adding Interrupt Support

In this section...
"Introduction" on page 5-7
"Guidelines for Creating Interrupt Functions" on page 5-9
"Filling in the Driver `board` Structure" on page 5-10

Introduction

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

- A descriptor file to connect a board type to the functions needed 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 5-15 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.

Function	Description
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 target 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 directory:

> *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 As necessary, create the interrupt functions the PreHook, PostHook, Start, and Stop functions and add them to the hook file. See "Guidelines for Creating Interrupt Functions" on page 5-9 for information on how to create these functions.

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

> *matlabroot*\toolbox\rtw\targets\xpc\target\build\
> xpcblocks\thirdpartydrivers

For example:

> *your_company_name*_int.m

The xPC Target software searches in this directory 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 xPC Target supported board for which interrupt functions have been written. See "Filling in the Driver board Structure" on page 5-10 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

xPC Target 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 5-15 for the prototypes for these functions.

To prepare for the creation of the hook file, examine the existing xPC Target 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 all 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 mdlStart routines.

  This function is typically used to enable interrupts from the board. The target 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 target 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 an appropriate board name string. The xPC Target 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 **xPC Target Options** section of the model Configuration Parameters dialog box

- `VendorId`, `DeviceId`, `SubVendorId`, `SubDeviceId` — Enter the appropriate ID strings. If you have a PCI board, the board manufacturer identifies

that board with either two or four ID values, depending on the specific hardware. When calling the hook functions, the xPC Target 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 xPC Target 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 communication driver.

- Enter the names of the interrupt functions. See "Hook Function Prototypes — Alphabetical List" on page 5-15 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 — A hook function for a PCI device has all fields of the xpcPCIDevice structure filled in except the *VirtAddress* field. To get the virtual address for a physical memory, in the Start function, call the xpcReserveMemoryRegion function and save the resulting virtual address in the *VirtAddress* field of the xpcPCIDevice structure. A pointer to the same instance of this structure is passed to all four functions. This action makes data, such as virtual addresses, available to all functions.

- ISA devices — A hook function for an ISA device has the base I/O address entered in the first physical address. No other fields in the xpcPCIDevice structure are filled in.

- `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

- *your_company_name_board*PostHook

- *your_company_name_board*PreHook

- *your_company_name_board*Start

- *your_company_name_board*Stop

*your_company_name_board*PostHook

Purpose	Run after return from interrupt service routine function-call subsystem or after sending wakeup call to model thread
Syntax	void __cdecl *your_company_name_board*PostHook(xpcPCIDevice *pciInfo*);
Argument	*pciInfo* Pointer to the PciDevice structure.
Description	*your_company_name_board*PostHook is not typically required. If you do not need this function, set it to 'NULL' in
	matlabroot\toolbox\rtw\targets\xpc\target\build\ xpcblocks\thirdpartydrivers*your_company_name*_int.m
See Also	xpcPCIDevice

Purpose	Run just before the interrupt service routine
Syntax	int __cdecl *your_company_name_board*PreHook(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

*your_company_name_board***Start**

Purpose	Run as the last item in mdlStart
Syntax	void __cdecl *your_company_name_board*Start(xpcPCIDevice **pciInfo*);
Argument	*pciInfo* Pointer to the PciDevice structure.
Description	*your_company_name_board*Start runs as the last item after all mdlStart functions. It is typically used to turn on interrupt generation.
See Also	xpcPCIDevice

Purpose   Run at the beginning of mdlTerminate

Syntax   void __cdecl *your_company_name_board*Stop(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 xPC Target Driver Notes

S-Function Guidelines

You implement xPC Target device driver blocks using Simulink S-functions. An S-function is a set of subroutines that implements a function. On the host, you can write an S-function in M-code, C, or Fortran. For xPC Target device drivers, you must write an S-function in C.

Simulink S-functions have a number of callback methods. For xPC Target 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. It also initializes hardware as necessary.
mdlOutputs	Computes the signals that this block emits.
mdlTerminate	Performs any actions required at termination of the simulation.

After you create the S-function, create a mask for it. See Chapter 4, "Masking Drivers". Also, refer to the Simulink documentation (in particular, *Writing S-Functions*).

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. All S-function work variables are cleared after calling mdlTerminate. This implies that each time the S-function calls mdlStart, you must reinitialize all work variables.

- Declare all memory-mapped registers as volatile.

- An S-function is compiled into a MEX-file to run as part of the simulated model on the host PC. 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 application to run on the target PC. Because of the following reasons, you must conditionally compile code for the host PC and the target PC.

- The host PC runs Windows and the target PC runs the xPC Target kernel.

- The host PC does not have the same I/O hardware as the target PC.

The preprocessor symbol MATLAB_MEX_FILE is defined when you compile for simulation (via mex). Undefine this symbol when compiling for the xPC Target environment. Use this symbol to conditionally compile host PC or target PC specific code. For example:

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

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

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

This provides definitions for the functions exported by the xPC Target kernel. The xPC Target kernel exports a number of functions for use in device drivers.

See "mdlStart and mdlTerminate Considerations" on page 6-4 for notes on specific applications of the callback methods.

mdlStart and mdlTerminate Considerations

When you load a target application onto a target PC, the driver executes the `mdlStart` callback method. If the execution is successful, the driver then executes `mdlTerminate`.

If `mdlStart` does not successfully complete, the application does not execute `mdlTerminate`. (Typically, `mdlStart` might not successfully complete if the application cannot find a referenced I/O board or if the board does not successfully initialize.)

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

With the above considerations, write `mdlStart` and `mdlTerminate` so that they cancel each other out. Ensure that `mdlTerminate` deallocates any resources that you 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 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 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" in *Writing 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 xPC Target I/O read and write functions:

- 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 address is the value returned in the BaseAddress array.

Memory-Mapped Space

For registers in memory-mapped space, access them by dereferencing through a pointer that contains the virtual address returned by the xpcReserveMemoryRegion function. Because modern compilers have aggressive optimizers, you must declare the pointer to be volatile. Doing so ensures that 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 hardware uses registers with different lengths, it might be easier to use the structure method.

- Array

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

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

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

7

Creating Custom Drivers Using the xPC Target Driver Authoring Tool

- "xPC Target Driver Authoring Tool" on page 7-2
- "Generating Custom Driver Templates" on page 7-4

xPC Target Driver Authoring Tool

xPC Target 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 xPC Target Driver Authoring Tool is not useful for these more complicated applications.

Based on the inputs you provide to xPC Target 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. Real-Time Workshop TLC code generation file. You typically need a .tlc file only if you want to inline your custom driver. See "Inlining xPC Target Drivers" on page 1-10 for further information. The xPC Target Driver Authoring Tool creates this file for you whether or not you want to inline the driver.
driver_block_name.mdl	Optional. Block mask model file for driver. After the xPC Target 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 xPC Target Driver Authoring Tool creates custom driver templates using the Legacy Code Tool (LCT). You do not need any prior knowledge of the Legacy Code Tool to use the xPC Target Driver Authoring Tool. If you want to read about the Legacy Code Tool, see "Integrating Existing C Functions into Simulink Models with the Legacy Code Tool" in *Writing S-Functions*.

Generating Custom Driver Templates

Using the xPC Target Driver Authoring Tool

The prerequisites for creating a custom xPC Target device driver using the xPC Target 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:
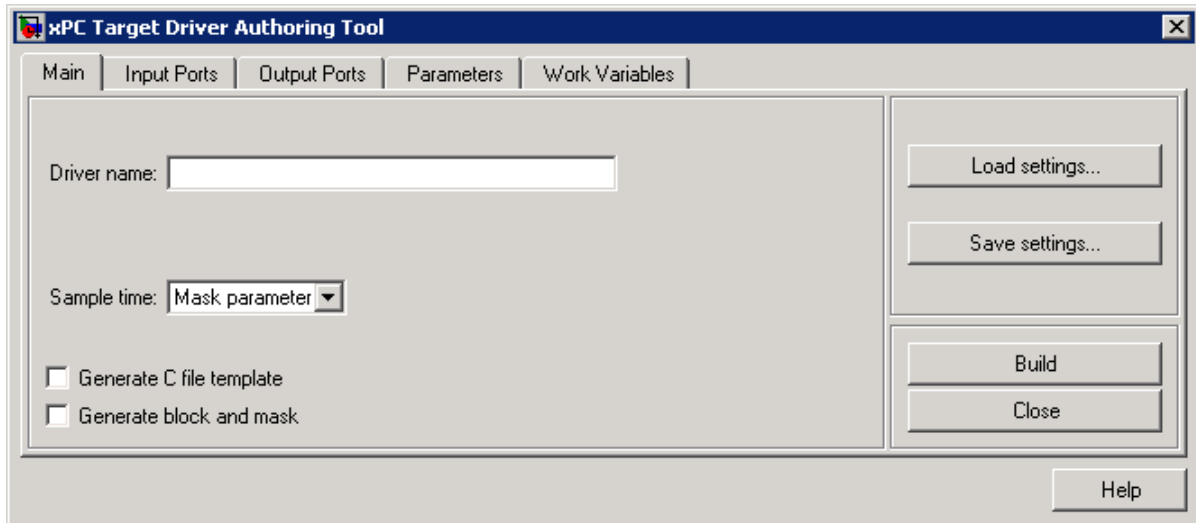
> Input ports
> Output ports
> Parameters
> Work variables

Setting Up Driver Variables

1 In the MATLAB Command Window, change directory to the one in which you want to save the driver code.

2 Start xPC Target Driver Authoring Tool. Type

    xpcdrivertool

The xPC Target 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 as necessary for your defined 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** — This check box is always selected. It ensures that the input port value will be passed 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 as necessary for your defined 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** — This check box is always selected. It ensures that the output port value will be passed 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 as necessary for your defined 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 as necessary for your defined 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 xPC Target Driver Authoring Tool allows you to save your configuration session as a MAT-file.

1 In the xPC Target Driver Authoring Tool, click the **Main** tab.

2 Click **Save settings**.

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

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

Reloading the Configuration

The xPC Target Driver Authoring Tool allows you to reload your configuration session as a MAT-file.

1 In the xPC Target 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 xPC Target 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 xPC Target Driver Authoring Tool or the mex function.

> **Note** Use the xPC Target 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 xPC Target 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 7-9).

Creating a C MEX File Using the xPC Target Authoring Tool

1 In the xPC Target 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 directory to the one that contains the driver files.

2 Compile and link the MEX-file. For example:

```
mex testdriver.c
```

This function creates the `sfcn_testdriver.win32mex` file.

Customizing the Device Driver Mask

The xPC Target Driver Authoring Tool creates a mask for the device driver. You can customize this block mask as described in Chapter 4, "Masking Drivers". If you customize the mask, do not use the xPC Target Driver Authoring Tool again to build your files. Doing so overwrites the driver files and you will lose your mask customizations.

8

I/O Structures — By Category

I/O Structures —
Alphabetical List

xpcPCIDevice

Purpose   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. As necessary, enter the return value from xpcReserveMemoryRegion. See "Description" on page 9-3 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 hardware manufacturer manual.

AddressSpaceIndicator	Indicates whether the board is I/O port mapped or memory-mapped. Values are one of the following. Verify this value in the hardware manufacturer manual for accuracy.	
	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

xpcPCIDevice

of these addresses. Base addresses are filled in during the BIOS plug and play initialization, before the xPC Target kernel starts to execute. The hardware 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 hardware manufacturer manual 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 all to a board access library.

See Also   `xpcGetPCIDeviceInfo`, `xpcShowPCIDeviceInfo`

Purpose   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

xpcTime

10

I/O Functions — By Category

Port I/O

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

PCI Configuration Information

xpcGetPCIDeviceInfo	Return information for PCI device
xpcShowPCIDeviceInfo	Display contents of PCIDevice structure

Physical Memory

xpcAllocPhysicalMemory	Allocate physical memory
xpcFreePhysicalMemory	Free physical memory
xpcReserveMemoryRegion	Return virtual address that corresponds to physical address and mark region as readable/writable

Time

xpcGetElapsedTime	Return time since system boot
xpcSubtractTime	Return difference between two times

Miscellaneous

`xpcBusyWait`	Wait for specified length of time in seconds
`xpcIsModelInit`	Return target application load state

I/O Functions — Alphabetical List

xpcAllocPhysicalMemory

Purpose	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 when absolutely necessary, such as for DMA transfers.
See Also	xpcFreePhysicalMemory

xpcBusyWait

Purpose   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

Purpose	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

Purpose   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. You can
                  pass a NULL pointer for the upTime argument if you do not want the
                  time in nanoseconds.

See Also   xpcTime, xpcSubtractTime

xpcGetPCIDeviceInfo

Purpose	Return information for PCI device
Prototype	int32_T xpcGetPCIDeviceInfo (uint16_T *vendorId*, uint16_T *deviceId*, uint16_T *subVendorId*, uint16_T *subDeviceId*, uint16_T *bus*, uint16_T *slot*, xpcPCIDevice **pciInfo*);

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 the subvendor or subdevice ID, or XPC_NO_BUS_SLOT for the device slot, the function will search through the PCI BIOS and return the first board that it finds with the specified IDs. If you specify a bus and a slot value, the function will successfully return only a board with the matching IDs found at that bus or slot.

You must supply valid vendor and device IDs. If you specify values other than XPC_NO_SUB for subvendor and subdevice IDs, the function will match 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 upon success. Otherwise, it returns a nonzero value.

See Also   xpcPCIDevice xpcShowPCIDeviceInfo

xpcInpB, xpcInpW, xpcInpDW

Purpose	I/O port input functions for byte, word, and double word accesses
Prototype	uint8_T xpcInpB(uint16_T *port*) uint16_T xpcInpW(uint16_T *port*) uint32_T xpcInpDW(uint16_T *port*)
Arguments	*port* Enter the port value
Header File	xpctarget.h
Description	These functions input data from the I/O port space. Use xpcInpB for byte access (8-bit), xpcInpW for word accesses (16-bit), and xpcInpDW for double word accesses (32-bit).
See Also	xpcOutpB, xpcOutpW, xpcOutpDW

xpcIsModelInit

Purpose   Return target 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 target application load state:

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

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

See Also   "mdlStart and mdlTerminate Considerations" on page 6-4

xpcOutpB, xpcOutpW, xpcOutpDW

Purpose	I/O port output functions for byte, word, and double word accesses

Prototype	`void xpcOutpB(uint16_T port, uint8_T value)`
	`void xpcOutpW(uint16_T port, uint16_T value)`
	`void xpcOutpDW(uint16_T port, uint32_T value)`

Arguments

port	Enter the port value
value	Contains the output value

Header File

xpctarget.h

Description   These functions output data to the I/O port space. Use `xpcOutpB` for byte access (8-bit), `xpcOutpW` for word accesses (16-bit), and `xpcOutpDW` for double word (16-bit) accesses.

See Also   xpcInpB, xpcInpW, xpcInpDW

Purpose   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. This is typically obtained 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.

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

xpcShowPCIDeviceInfo

Purpose	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 deploying.

See Also	xpcGetPCIDeviceInfo

xpcSubtractTime

Purpose   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. You can
                pass a NULL pointer for the time argument if you do not want the
                time in nanoseconds.

See Also   xpcTime, xpcGetElapsedTime