# xPC Target

## For Use with Real-Time Workshop®

- Modeling
- Simulation
- Implementation

User's Guide

*Version 2*

**How to Contact The MathWorks:**

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

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

508-647-7000 Phone

508-647-7001 Fax

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

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

*xPC Target User's Guide*



Printing History:

| | | |
|---|---|---|
| September 1999 | First printing | New for Version 1 (Release 11.1) |
| November 2000 | Online only | Revised for Version 1.1 (Release 12) |
| June 2001 | Online only | Revised for Version 1.2 (Release 12.1) |
| September 2001 | Online only | Revised for Version 1.3 (Release 12.1+) |
| July 2002 | Online only | Revised for Version 2 (Release 13) |
| November 2003 | Online only | Revised for Version 2.0.2 (Release 13 SP1) |
| October 2004 | Online only | Revised for Version 2.0.3 (Release 13SP2) |

# Contents

## 1 Advanced Topics

## 2 Graphical User Interfaces



## 3 Embedded Option

## 4 Software Environment

## 5 Target Objects



## 6 Scope Objects



## Index

1

# Advanced Topics

After learning the basic procedures for creating and running a target application, acquiring signal data and tuning parameters, you can try some of the special and advanced procedures with xPC Target. This chapter includes the following sections:

| | |
|---|---|
| I/O Driver Blocks (p. 1-2) | Add I/O driver blocks to your Simulink® model to connect your model to sensors and actuators |
| Polling Mode (p. 1-19) | Use polling mode as an alternative to interrupt mode for reducing latency times with I/O drivers |
| Target PC Command-Line Interface (p. 1-31) | Enter commands on the target PC for stand-alone applications that are not connected to the host PC |
| Web Browser Interface (p. 1-40) | Connect a target application running on a target PC to any host PC connected to a network |

# I/O Driver Blocks

You add I/O driver blocks to your Simulink model to connect your model to physical I/O boards. These I/O boards then connect to the sensors and actuators in the physical system. This section includes the following topics:

- I/O Driver Block Library
- Memory-Mapped Devices
- PCI Bus I/O Devices
- xPC Target I/O Driver Structures
- Updated Driver Information
- Adding I/O Blocks with the xPC Target Library
- Adding I/O Blocks with the Simulink Library Browser
- Defining I/O Block Parameters

## I/O Driver Block Library

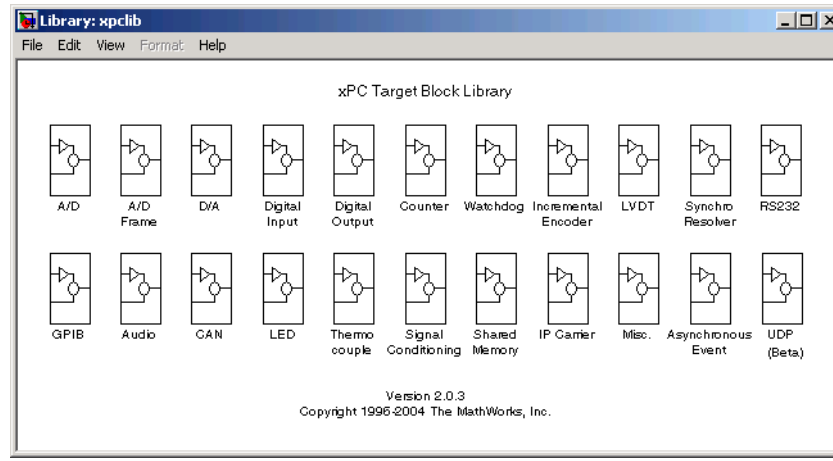
A driver block does not represent an entire board, but an I/O section supported by a board. Therefore, the xPC Target library can have more than one block for each physical board. I/O driver blocks are written as C-code S-functions (noninlined S-functions). The source code for the C-code S-functions with xPC Target is included.

xPC Target supports PCI and ISA buses. If the bus type is not indicated in the driver block number, you can determine the bus type of a driver block by checking the block's parameter dialog box. The last parameter is either a PCI slot, for PCI boards, or a base address, for ISA boards.

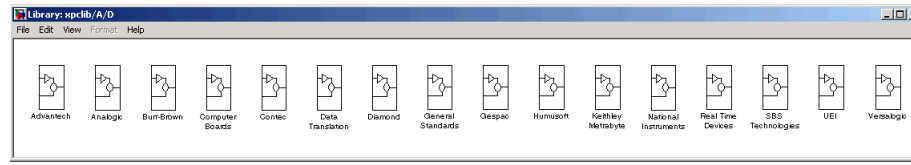
You can open the I/O device driver library with the MATLAB® command `xpclib`. The library `xpclib` contains sublibraries grouped by the type of I/O function they provide.

**Note** Opening a dialog box for a source block causes Simulink to pause. While Simulink is paused, you can edit the parameter values. You must close the dialog box to have the changes take effect and allow Simulink to continue.

In particular, hardware input blocks in the xPC Target library (blocks that acquire data from hardware) are affected by this change.

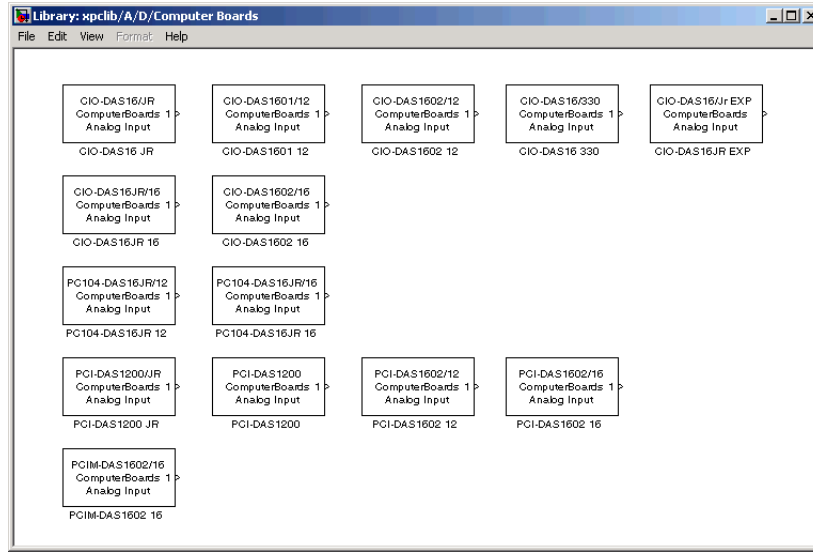



When you double-click one of these groups, the sublibrary opens, displaying a list grouped by manufacturer as shown below.




Double-clicking one of the manufacturer groups then displays the set of I/O device driver blocks for the specified I/O functionality (for example, A/D, D/A, Digital Inputs, Digital Outputs, and so on).

The following figure shows the A/D drivers for the manufacturer ComputerBoards, Inc.




When you double-click one of these blocks, a **Block Parameters** dialog box opens, allowing you to make hardware-specific parameters. Parameters typically include

- Sample time
- Number of channels
- Voltage range
- Base address (ISA boards)

## Memory-Mapped Devices

Some supported boards in the xPC Target I/O library are memory-mapped devices, for example, Burr-Brown boards. These memory-mapped boards are accessed in the address space between 640 K and 1 M in the lower memory area. xPC Target reserves a 112 kB memory space for memory-mapped devices in the address range

```
C0000 - DC000
```

Base addresses of memory-mapped devices must be chosen within this memory space for your target application to work properly.

## ISA Bus I/O Devices

There are two types of ISA boards in the market:

- Jumper addressable ISA cards
- PnP (Plug and Play) ISA cards

xPC Target only supports jumper addressable ISA cards (non-PnP ISA boards) where you have to set the base address manually.

## PCI Bus I/O Devices

The xPC Target I/O library supports I/O boards with a PCI bus. During the boot process, the BIOS creates a conflict-free configuration of base addresses and interrupt lines for all PCI devices in the target system. The user does not need to define any base address information in the dialog boxes of the drivers.

All PCI device driver blocks have an additional entry in their dialog boxes. This entry is called `PCI Slot (-1 Autodetect)` and allows you to use several identical PCI boards within one target system. This entry uses a default value of `-1`, which allows the driver to search the entire PCI bus to find the board. When more than one board of the same type is found, it is necessary for you to use a designated slot number and avoid the use of autodetection. For manually setting the slot number you use a number greater than or equal to 0. If the board is not able to locate this slot in the target PC, your target application will generate an error message after downloading.

If this additional entry is set to any value equal to or greater than `0`, you must be aware of the manufacturer's identification number (Vendor ID) and the board identification number (Device ID) of those boards supported by the I/O library. When the target is booted, the BIOS is executed and the target PC monitor shows parameters for any PCI boards installed on the target PC. An example is shown below:

| Bus No | Device No. | Func. No. | Vendor ID | Device ID | Device Class | IRQ |
|---|---|---|---|---|---|---|
| 0 | 4 | 1 | 8086 | 7111 | IDE controller | 14/15 |
| 0 | 4 | 2 | 8086 | 7112 | Serial bus controller | 10 |
| 0 | 11 | 0 | 1307 | 000B | Unknown PCI device | N/A |
| 1 | 0 | 0 | 12D2 | 0018 | Display controller | 11 |

In this example, the third line indicates the location of the ComputerBoards PCI-DIO48 board. This is known since the ComputerBoards Vendor ID is `0x1307` and the Device ID is `0xb`. In this case, you now can see that the ComputerBoards board is plugged into the PCI slot 11 (Device No.), and that this value must be entered in the dialog box entry in your I/O device driver for each model that uses this I/O device.

## xPC Target I/O Driver Structures

Properties for xPC Target I/O drivers are usually defined using the parameter dialog box associated with each Simulink block. However, for more advanced drivers, the available fields defined by text boxes, check boxes, and pull-down lists are inadequate to define the behavior of the driver. In such cases, a more textual description is needed to indicate what the driver has to do at run-time. *Textual* in this context refers to a programming language like syntax and style.

xPC Target currently uses a textual description contained in message structures for the RS-232, GPIB, CAN (initialization), and the general counter drivers (AMD9513).

**What is a message structure?** — A message structure is a MATLAB array with each cell containing one complete message (command). A message consists of one or more statements.

| First message | Second message | Third message |
|---|---|---|
| Message(1).field | Message(1).field | Message(1).field |
| Message(1).field | Message(1).field | Message(1).field |
| Message(1).field | Message(1).field | Message(1).field |

**Syntax of a message statement** — Each statement in a message has the following format:

```
Structure_name(index).field_name = <field string or value>
```

The field names are defined by the driver, and need to be entered with the correct upper- and lowercase letters. However, you can choose your own structure name and enter that name into the driver parameter dialog box.

**Creating a message structure** — You could enter the message structure directly in the edit field of the driver parameter dialog box. But because the message structure is an array and very large, this becomes cumbersome very easily.

A better way is to define the message structure as an array in an M-file and pass the structure array to the driver by referencing it by name. For example, to initialize an external A/D module and acquire a value during each sample interval, create an M-file with the following statements:

```
Message(1).senddata='InitADConv, Channel %d'
Message(1).inputports=[1]
Message(1).recdata=''
Message(1).outputports=[]

Message(2).senddata='Wait and Read converted Value'
Message(2).inputports=[]
Message(2).recdata='%f'
Message(2).outputports=[1]
```

This approach is different from other xPC Target driver blocks:

- The M-file containing the definition of the message structure has to be executed before the model is opened.

  After creating your Simulink model and message M-file, set the preload function of the Simulink model to load the M-file the next time you open the model. In the MATLAB window, type

  ```
  set_param(gcs, 'PreLoadFcn', 'M-file_name'
  ```

- When you move or copy the model file to a new directory, you also need to move or copy the M-file defining the message structure.

During each sample interval, the driver block locates the structure defined in the **Block Parameters** dialog box, interprets the series of messages, and executes the command defined by each message.

**Specific drivers and structures** — For detailed information on the fields in a message structure, see the following chapters in the xPC Target I/O Reference documentation:

- Chapter 1, “Serial Communications Support”
- Chapter 2, “GPIB I/O Support”
- Chapter 3, “CAN I/O Support”

## Updated Driver Information

Since we are always updating and adding new drivers to xPC Target, not all of the information about these drivers is included in the online or printed documentation.

For updated and additional driver information, see our developer Web site at

```
http://www.mathworks.com/support/product/XP/productnews/
productnews.shtml
```

## Adding I/O Blocks with the xPC Target Library

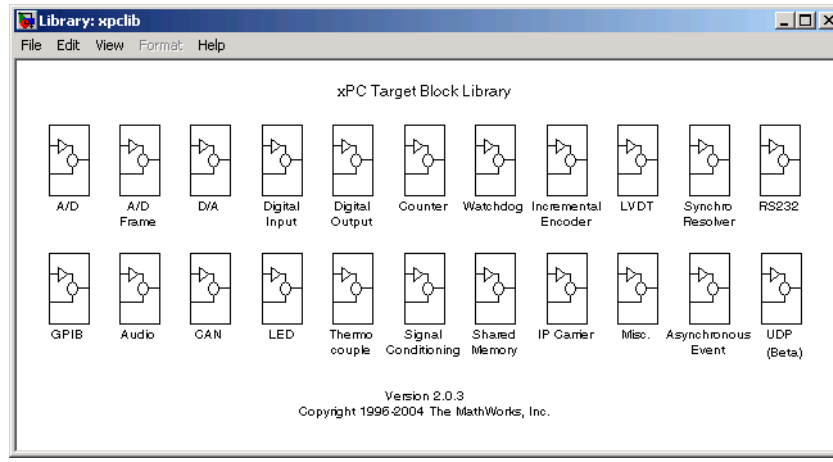
xPC Target includes a Simulink block library for I/O drivers. The highest hierarchical level in the library is grouped by I/O function. The second level is grouped by board manufacturer. The manufacturer groups within this second level contain the driver blocks for specific boards.

This procedure uses the Simulink model `xpc_osc.mdl` as an example of how to add and connect I/O blocks.

**1** In the MATLAB window, type

```
xpclib
```

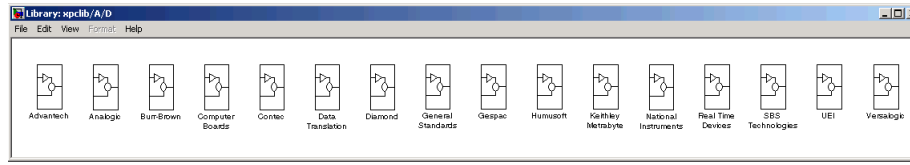
The **Library: xpclib** window opens.




Alternatively, you can access the I/O driver library with the Simulink Library Browser. See “Adding I/O Blocks with the Simulink Library Browser” on page 1-12.

**2** Open a function group. For example, to open the A/D group, double-click the **A/D** block.

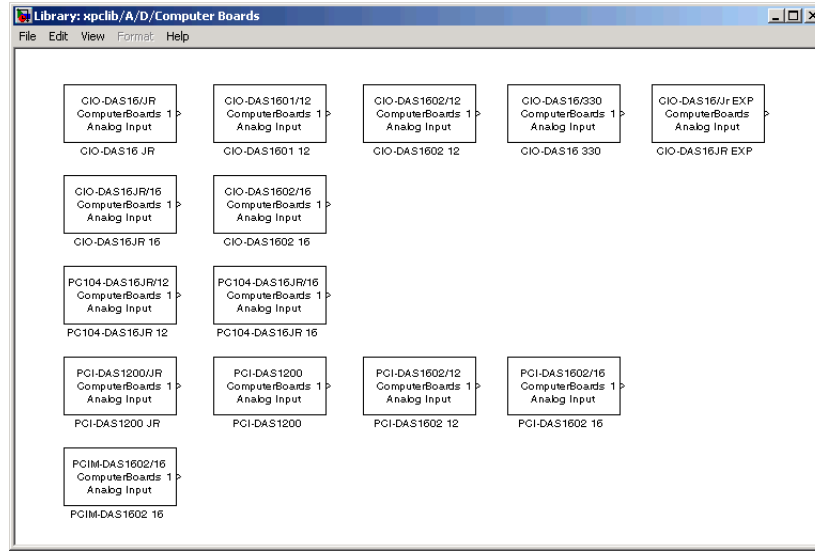
The manufacturer level opens.




Within each manufacturer group are the blocks for a single function.

**3** Open a manufacturer group. For example, to open the A/D driver blocks from ComputerBoards, double-click the group marked **ComputerBoards**.

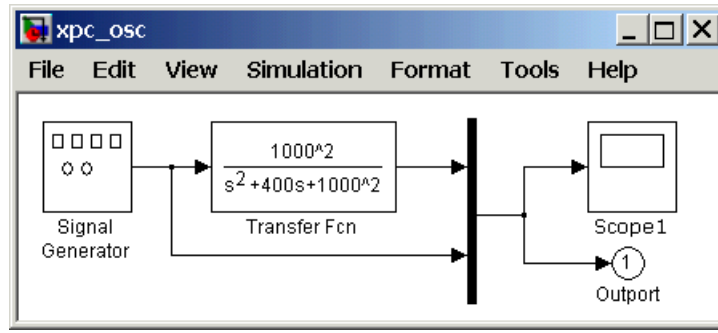
The window with the A/D driver blocks for ComputerBoards opens.




**4** In the Simulink window, type

```
xpc_osc
```

The Simulink block diagram opens for the model `xpc_osc.mdl`.

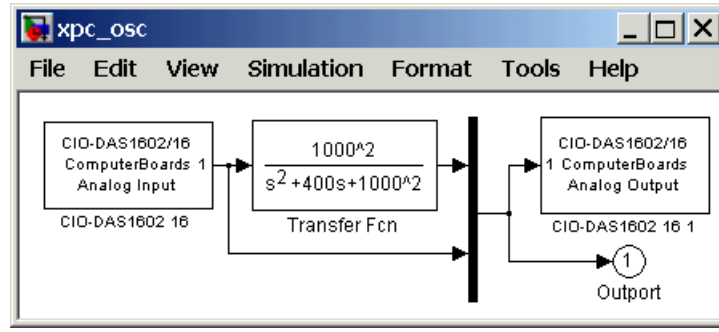



5 From the block library, click-and-drag the name of an A/D board to the Simulink block diagram. Likewise, click-and-drag the name of a D/A board to your model.

Simulink adds the new I/O blocks to your model.

6 Remove the Signal Generator block and add the Analog Input block in its place. Remove the Scope block and add the Analog Output block in its place.

The demo model `xpcosc` should look like the figure shown below.




**Note** You cannot run this model unless you have the I/O board shown installed in your target PC. However, you can substitute the driver blocks for another I/O board that is installed in the target PC.

Your next task is to define the I/O block parameters. See “Defining I/O Block Parameters” on page 1-16.

## Adding I/O Blocks with the Simulink Library Browser

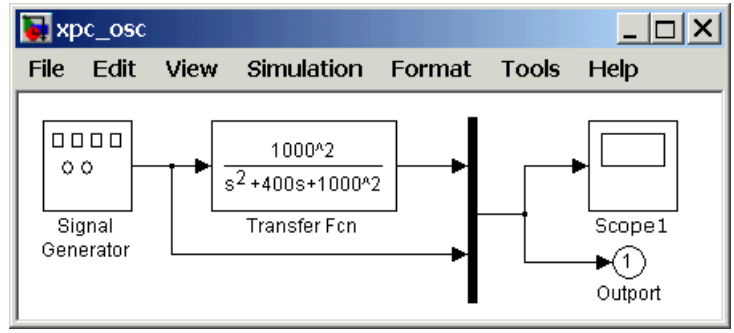
xPC Target includes a Simulink block library for I/O drivers. The highest hierarchical level in the library is grouped by I/O function. The second level is grouped by board manufacturer. The manufacturer groups within this second level contain the driver blocks for specific boards.

This procedure uses the Simulink model `xpc_osc.mdl` as an example of how to add and connect I/O blocks.

1 In the MATLAB window, type

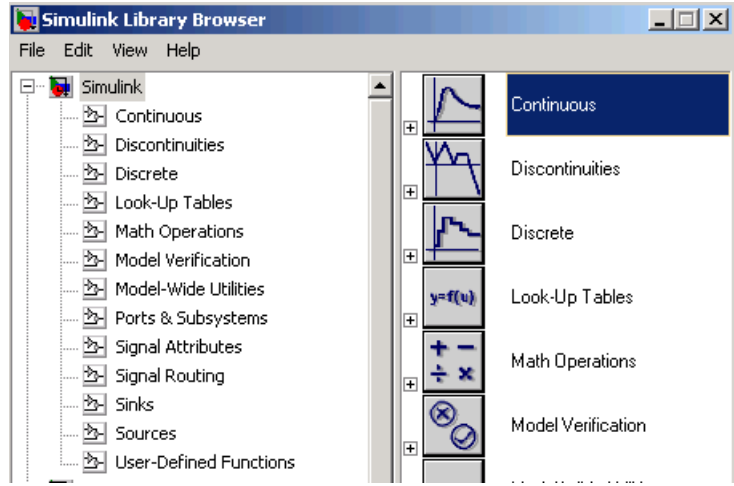
```
xpc_osc
```

The Simulink block diagram opens for the model `xpc_osc.mdl`.




2 In the Simulink window, from the **View** menu, click **Show Library Browser**.

The Simulink Library Browser window opens. Alternatively, you can open the Simulink Library Browser by typing `simulink` in the MATLAB Command Window.

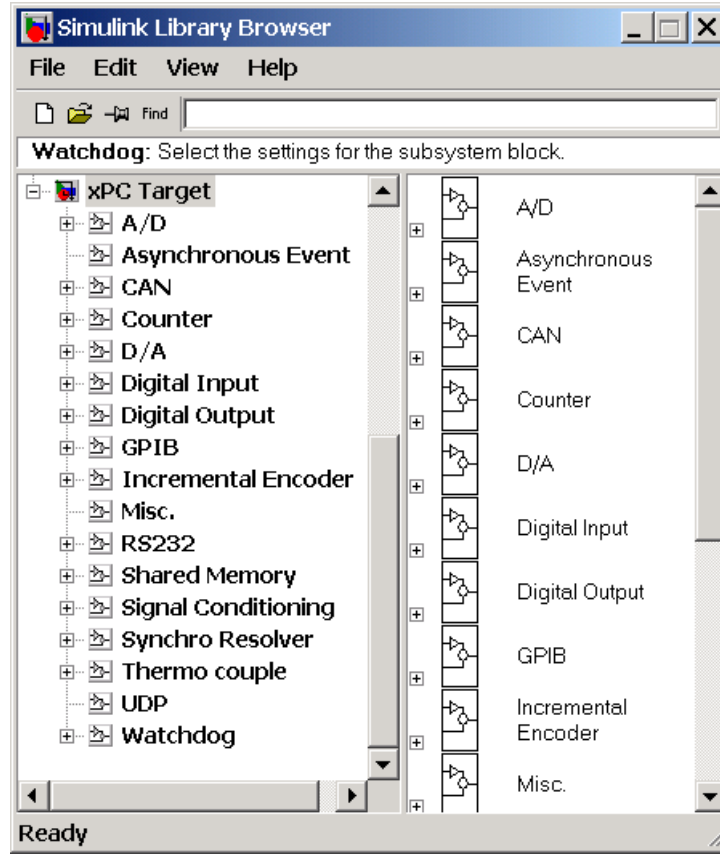



You can access the xPC Target I/O library by right-clicking **xPC Target**, and then clicking **Open the xPC Target Library**.

Alternatively, you can access driver blocks using the xPC Target I/O driver library. See “Adding I/O Blocks with the xPC Target Library” on page 1-9.

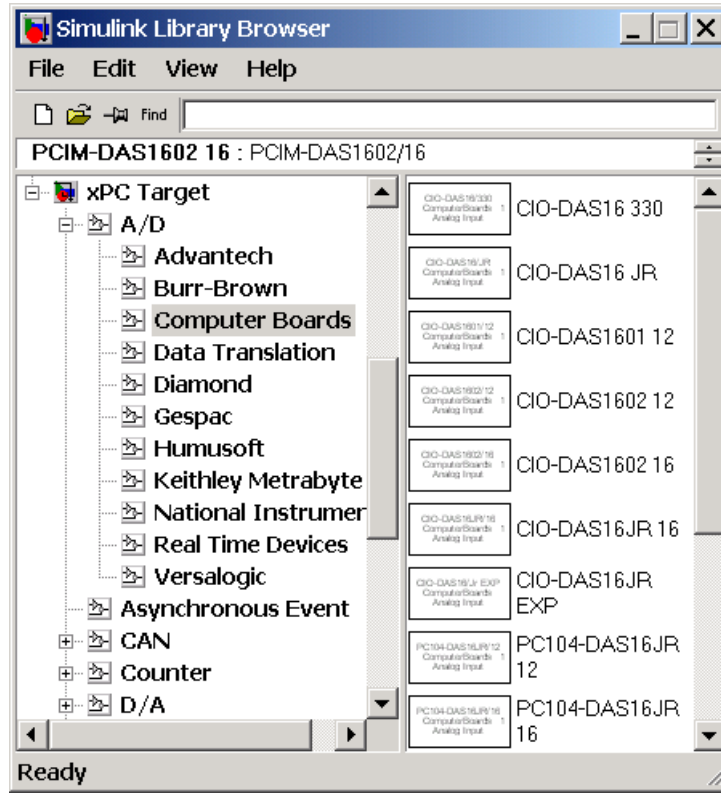
**3** Double-click **xPC Target**.

A list of I/O functions opens.




4 Open a function group. For example, to open the A/D group for ComputerBoards, double-click **A/D**, and then double-click **ComputerBoards**.

A list with the A/D driver blocks for ComputerBoards opens.

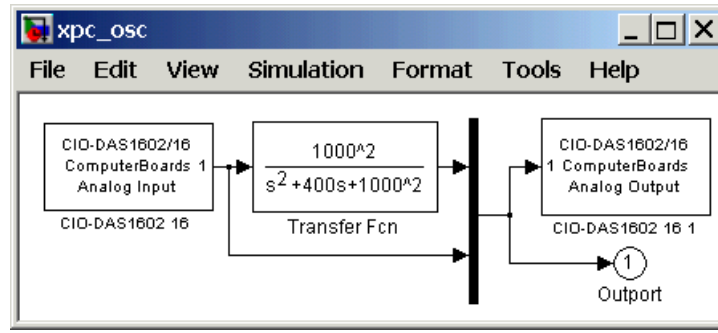



5 From the block library, click-and-drag the name of an A/D board to the Simulink block diagram. Likewise, click-and-drag the name of a D/A board to your model.

   Simulink adds the new I/O blocks to your model.

6 Remove the Signal Generator block and add the analog input block in its place. Remove the Scope block and add the analog output block in its place.

   The model `xpcosc` should look like the figure shown below.




**Note** You cannot run this model unless you have the I/O board shown above installed in your target PC. However, you can substitute the driver blocks for another I/O board that is installed in the target PC.

Your next task is to define the I/O block parameters. See "Defining I/O Block Parameters".

## Defining I/O Block Parameters

The I/O block parameters define values for your physical I/O boards. For example, I/O block parameters include channel numbers for multichannel boards, input and output voltage ranges, and sample time.

This procedure uses the Simulink model `xpc_osc.mdl` as an example, and assumes you have added an analog input and an analog output block to your model. To add an I/O block, see either "Adding I/O Blocks with the xPC Target Library" on page 1-9 or "Adding I/O Blocks with the Simulink Library Browser" on page 1-12.

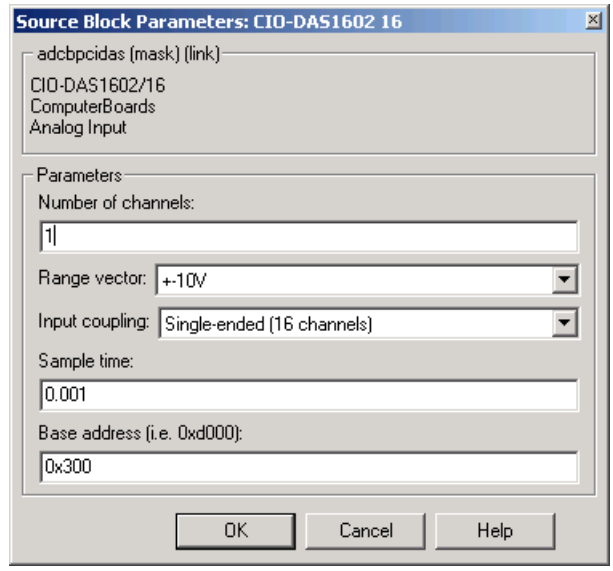
1 In the Simulink window, double-click the input block labeled **Analog Input**.

  The dialog box for the A/D converter opens.

2 Fill in the dialog box. For example, for a single channel enter `1` in the **Number of Channels** box, select `–10 V` for the input range, and select `single-ended (16 channels)` for the MUX switch position. Enter the same

sample time you entered for the step size in the **Simulation Parameters** dialog box. Enter the base address for this ISA-bus board.

The **Block Parameters** dialog box should look similar to the figure shown below.

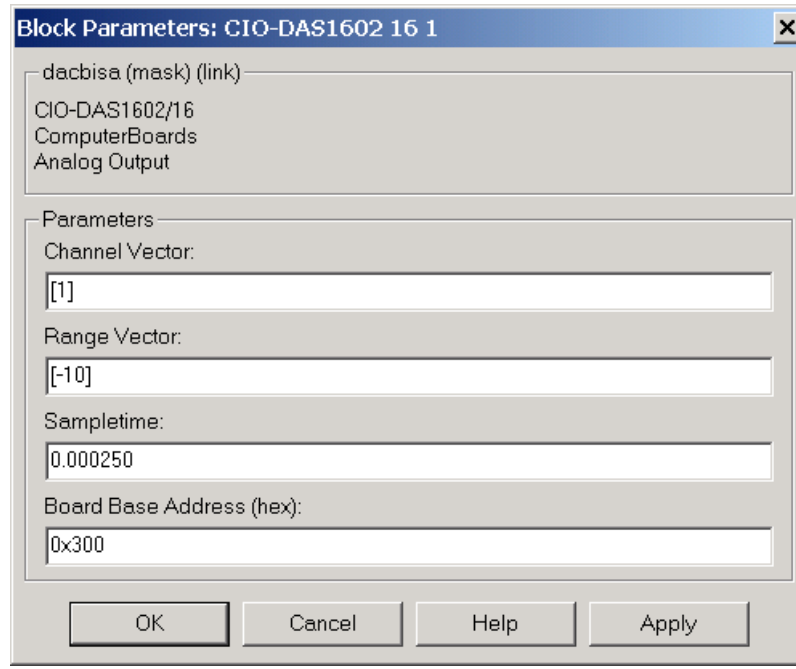



**3** In the Simulink window, double-click the output block labeled **Analog Output**.

The dialog box for the D/A converter opens.

**4** Fill in the dialog box. For example, for one channel enter [1] in the **Channel Vector** box; for an output level of –10 V enter the code [-10] in the **Range Vector** box. Enter the same sample time you entered for the step size in the **Simulation Parameters** dialog box. Enter the base address for this ISA-bus board.

The **Block Parameters** dialog box should look similar to the figure shown below.




If you change the sample time by changing the target object property `SampleTime`, the sample times you entered in both of the I/O blocks are set to the new value. The step size you entered in the **Simulation Parameters** dialog box remains unchanged.

Your next task is to build and run the target application. See “xPC Target Application” on page 3-24.

# Polling Mode

A good understanding of polling mode will help you to effectively use it, and a better understanding of interrupt mode will help you to decide under which circumstances it makes sense for you to switch to the polling mode. See “Interrupt Mode” on page 1-19 and “Polling Mode” on page 1-21. This section includes the following topics:

- xPC Target Kernel Polling Mode
- Interrupt Mode
- Polling Mode
- Setting the Polling Mode
- Restrictions Introduced by the Polling Mode
- Controlling the Target Application
- Polling Mode Performance

## xPC Target Kernel Polling Mode

Polling mode for the xPC Target real-time kernel is designed to execute target applications at sample times close to the limit of the hardware (CPU). Using polling mode with high-speed and low-latency I/O boards and drivers, allows you to achieve smaller sample times for applications that you cannot achieve using the interrupt mode of xPC Target.

Polling mode has two main applications:

- **Control applications** — Control applications of average model size and I/O complexity that are executed at very small sample times (Ts = 5 to 50 us)
- **DSP applications** — Sample-based DSP applications (mainly audio and speech) of average model size and I/O complexity that are executed at very high sample rates (Fs = 20 to 200 kHz)

## Interrupt Mode

Interrupt mode is the default real-time execution mode for the xPC Target kernel. This mode provides the greatest flexibility and is the mode you should choose for any application that executes at the given base sample time without overloading the CPU.

The scheduler assures real-time single-tasking and multitasking execution of single-rate or multirate systems including asynchronous events (interrupts). Additionally, background tasks like host-target communication or updating the target screen run in parallel with the sample-time-based model tasks. This allows you to interact with the target system while the target application is executing in real time at high sample rates. This is made possible by an interrupt-driven real-time scheduler that is responsible for executing the various tasks according to their priority. The base sample time task can interrupt any other task (larger sample time tasks or background tasks) and execution of the interrupted tasks resumes as soon as the base sample time task completes operation. This gives a quasi-parallel execution scheme with consideration to the priorities of the tasks.

### Latencies Introduced by Interrupt Mode

Compared to other modes, interrupt mode has more advantages. The exception is the disadvantage of introducing a constant overhead or latency, which reduces the minimal possible base sample time to a constant number. The overhead is the sum of various factors related to the interrupt-driven execution scheme and can be referred to as overall interrupt latency. The overall latency consists of the following parts, assuming that the currently executing task is not executing a critical section and has therefore not disabled any interrupt sources.

- **Interrupt controller latency** — In a PC-compatible system the interrupt controller is not part of the x86-compatible CPU but part of the CPU chipset. The controller is accessed over the I/O-port address space, which introduces a read or write latency of about 1 us for each 8 bit/16 bit register access. Because the CPU has to check for the interrupt line requesting an interrupt, and the controller has to be reset after the interrupt has been serviced, a latency of about 5 us is introduced to properly handle the interrupt controller.
- **CPU hardware latency** — Modern CPUs try to predict the next couple of instructions, including branches, by the use of instruction pipelines. If an interrupt occurs, the prediction fails and the pipeline has to be fully reloaded. This process introduces an additional latency. Additionally, because of interrupts, cache misses will occur.
- **Interrupt handler entry and exit latency** — Because an interrupt can stop the currently executing task at any instruction and the interrupted task has to resume proper execution when the interrupting task completes execution,

its state has to be saved and restored accordingly. This includes saving CPU data and address registers, including the stack pointer. In the case that the interrupted task executed floating-point unit (FPU) operations, the FPU stack has to be saved as well (108 bytes on a Pentium CPU). This introduces additionally latency.

- **Interrupt handler content latency** — If a background task has been executing for a longer time, say in a loop, its needed data will be available in the cache. But as soon as an interrupt occurs and the interrupt service handler is executed, the data needed in the interrupt handler might no longer be in the cache, causing the CPU to reload it from slower RAM. This introduces additional latency. Generally, an interrupt reduces the optimal execution speed or introduces latency, because of its unpredictable nature.

The xPC Target real-time kernel in interrupt mode is close to optimal for executing code on a PC-compatible system. However, interrupt mode introduces an overall latency of about 8 us. This is a significant amount of time when considering that a 1 GHz CPU can execute thousands of instructions within 8 us. This time is equivalent to a Simulink model containing a hundred nontrivial blocks. Additionally, because lower priority tasks have to be serviced as well, a certain amount of headroom (at least 5%) is necessary, which can cause additional cache misses and therefore nonoptimal execution speed.

## Polling Mode

The polling mode for the xPC Target real-time kernel does not have the 8 us of latency that is with the interrupt mode. This is because the kernel does not allow interrupts at all, so the CPU can use this extra time for executing model code.

Polling mode is sometimes seen as a “primitive” or “brute force” real-time execution scheme. Nevertheless, when a real-time application executes at a given base sample time in interrupt mode and overloads the CPU, switching to polling mode is often the only alternative to getting the application executing at the required sample time.

*Polling* means that the kernel waits in an empty while loop until the time at which the next model step has to be executed is reached. Then the next model step is executed. For this at least a counter implemented in hardware has to be accessible by the kernel in order to get a base reference for when the next model step execution has to commence. The kernel polls this hardware counter. If this

hardware counter must be outside the CPU, say in the chipset or even on an ISA or PCI board, the counter value can only be retrieved by an I/O or memory access cycle that again introduces latency. This latency usually eats up the freed-up time of polling mode. Fortunately, since the introduction of the Pentium CPU family from Intel, the CPU is equipped with a 64 bit counter on the CPU substrate itself, which commences counting at power-up time and counts up driven by the actual clock rate of the CPU. Even a highly clocked CPU is not likely to lead to an overflow of a 64 bit counter (2^64 * 1e-9 (1 GHz CPU) = 584 years). All in all the Pentium counter comes with the following features:

- **Accurate measurements** — Because the counter counts up with the CPU clock rate (~1 GHz nowadays), the accuracy of time measurements even in the microsecond range is very high, therefore leading to very small absolute real-time errors.
- **No overflow** — Because the counter is 64 bits wide, in a practical application and overflow does not occur, which makes a CPU time expensive overflow handler unnecessary.
- **No latency** — The counter resides on the CPU. Reading the counter value can be done within one CPU cycle, introducing almost no latency.

The polling execution scheme is not dependent on any interrupt source to notify the code to continue calculating the next model step. While this frees up the CPU, it means that any code that is part of the exclusively running polling loop is executed in real time, even components, which have so far been executed in background tasks. Because these background tasks are usually non-real-time tasks and can use a lot of CPU time, there is only one solution: Do not execute them. And this is the big disadvantage of polling mode: In order to be efficient, only the target application's relevant parts should be executed, and this is, in the case of xPC Target, the code representing the Simulink model itself. Therefore, host-target communication and target display updating are disabled. Because polling mode reduces the commonly known features of xPC Target to a minimum, you should choose it only as the last possible alternative to reach the required base sample time for a given model. Therefore, the following should be assured before you consider polling mode.

- **The model is optimal concerning execution speed** — First, you should run the model through the Simulink profiler to find any possible speed optimizations using alternative blocks. If the model contains continuous states, the discretization of these states will reduce model complexity

significantly, because a costly fixed-step integration algorithm can be avoided. If continuous states cannot be discretized, the integration algorithm with the lowest order that still produces correct numerical results should be used.

- **Use the fastest available computer hardware** — Ensure that the CPU with the highest clock rate available is used for a given PC form factor. For the desktop form factor, this would mean a clock rate above 1 GHz; for a mobile application, say using the PC/104 form factor, this would mean a clock rate above 400 MHz. Most of the time, you should use a desktop PC, because the highest clocked CPUs are available for this form factor only. Executing `xpcbench` at the MATLAB command prompt gives an understanding about the best performing CPUs for xPC Target applications.
- **Use the lowest latency I/O hardware and drivers available** — Many xPC Target applications communicate with hardware through I/O hardware over either an ISA or PCI bus. Because each register access to such I/O hardware introduces a comparably high latency time (~1 us), the use of the lowest latency hardware/driver technology available is crucial.
- **The base sample time is about 50 us or less** — The time additionally assigned to model code execution in polling mode is only about 8 us. If the given base sample time of the target application exceeds about 50 us, the possible percentage gain is rather small. Other optimization technologies might have a bigger impact on increasing performance.

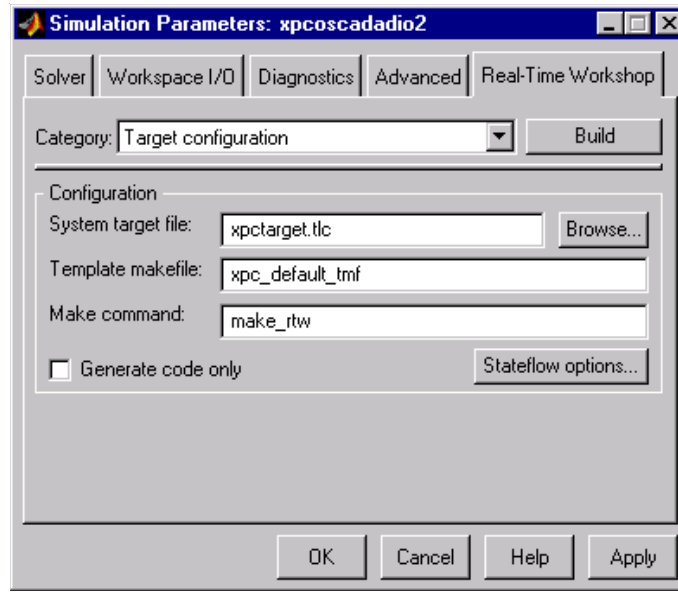
## Setting the Polling Mode

Polling mode is an alternative to the default interrupt mode of the real-time kernel. This means that the kernel on the bootable floppy disk created the `xpcsetup` GUI allows running the target application in both modes without the necessity to use another boot floppy disk.

By default the target application executes in interrupt mode. In order to switch to polling mode, you need to pass an option to the **System target file** command.

1. In the Simulink window, and from the **Tools** menu, point to **Real-Time Workshop**, and then click **Options**.

   The Simulation Parameters dialog box opens with the Real-Time Workshop® pane open.



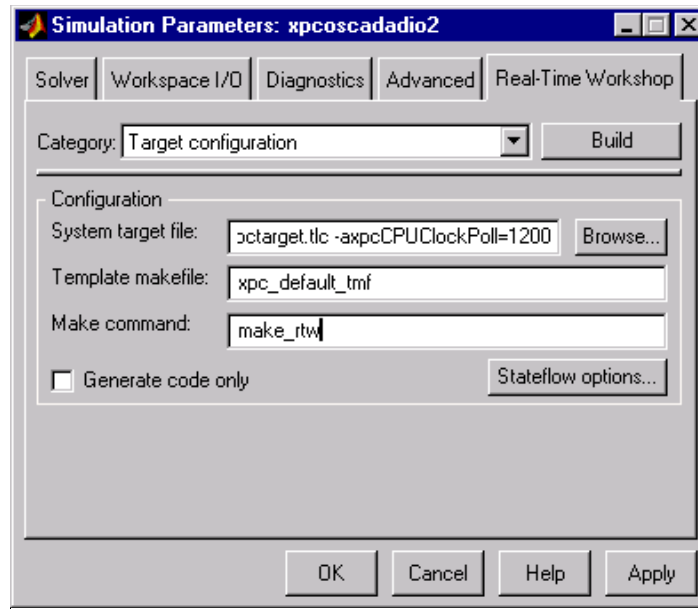

**2** In the **System target file** edit field, specify the option

```
-axpcCPUClockPoll=CPUClockRateMHz
```

The assignment of the clock rate of the target PC's CPU is necessary because the Pentium's on-chip counter used for polling mode counts up with the CPU clock rate. If the clock rate is provided, the kernel knows how to convert clock ticks to seconds and vice versa. If an incorrect clock rate is provided, the target application executes at an incorrect base sample time. You can find out about the CPU clock rate of the target PC by rebooting the target PC and checking the screen output during BIOS execution time. The BIOS usually displays the CPU clock rate in MHz right after the target PC has been powered up.

For example, if your target PC is a 1.2 GHz AMD Athlon, specify option

```
-axpcCPUClockPoll=1200
```




If you want to execute the target application in interrupt mode again, either remove the option or assign a CPU clock rate of 0 to the option:

```
-axpcCPUClockPoll=0
```

If you make a change to the **System target file** field, you need to rebuild the target application for the change to take effect. Building the target application, downloading it, and preparing it for a run then works exactly the same way as it did with default interrupt mode.

After the download of the target application has succeeded, the target screen displays the mode, and if polling mode is activated, it additionally displays the defined CPU clock rate in MHz. This allows checking for the correct setting.

## Restrictions Introduced by the Polling Mode

As explained above, polling mode executes the Simulink-based target application in real time exclusively. While the target application is executing in polling mode, the background tasks, mainly the one for host-target communication and target screen updating, are inactive. This is because all

interrupts of the target PC are fully disabled during the execution of the target application. On one hand this assures the highest polling performance; on the other hand, as a consequence the background tasks are not serviced.

Here is a list of all relevant restrictions of polling mode, which are otherwise available in the default interrupt mode.

### Host-Target Communication Is Not Available During the Execution of the Target Application

If the target application execution is started in polling mode, say with

```
start(tg)
```

host-target communication is disabled throughout the entire run, or in other words until the stop time is reached. Each attempt to issue a command like

```
tg
```

leads to a communication-related error message. Even the `start(tg)` command to start polling mode execution returns such an error message, because the host side does not receive the acknowledgment from the target before timing out. The error message when executing `start(tg)` is not avoidable. Subsequently, during the entire run, it is best not to issue any target-related commands on the host, in order to avoid displaying the same error message over and over again.

As a consequence, it is not possible to issue a `stop(tg)` command to stop the target application execution from the host side. The target application has to reach its set stop time in order that polling mode is exited. You can use

```
tg.stoptime=x
```

before starting the execution, but once started the application executes until the stop time is reached. Nevertheless, there is a way to stop the execution interactively before reaching the target application stop time. See “Controlling the Target Application” on page 1-29.

If the target application execution finally reaches the stop time and polling mode execution is stopped, host-target communication will begin functioning again. However, the host-target communication link might be in a bad state. If you still get communication error messages after polling mode execution stops, type the command

```
xpctargetping
```

to reset the host-target communication link.

After the communication link is working again, type

```
tg
```

in order to resync the target object on the host side with the most current status of the target application.

### Target Screen Does Not Update During the Execution of the Target Application

As with the restriction mentioned above, the target screen updating is disabled during the entire execution of the target application. Using the kernel with the **TargetScope** option enabled (see `xpcsetup`-GUI) loses its entire functionality during a run, and it is therefore recommended that you use the kernel with the **TargetScope** property disabled (text output only). The text mode enabled kernel actually provides more information when running in polling mode.

### Session Time Does Not Advance During the Execution of the Target Application

Because all interrupts are disabled during a run, the session time does not advance. The session time right before and after the run is therefore the same. This is a minor restriction and should not pose a problem.

### The Only Rapid-Prototyping Feature Available Is Data Logging

Because host-target communication and target screen updating are disabled during the entire run, most of the common rapid-prototyping features of xPC Target are not available in polling mode. These are

- Parameter tuning — Neither through the command-line interface nor through External mode
- Signal tracing through scope objects — Neither through scope objects of type `host` (`xpcscope`-GUI or scripts) or type `target` (scopes on the target screen if property **TargetScope** is enabled)
- Signal monitoring — You cannot run a GUI interface on the host PC using an environment that depends on communication between the host and target computers
- Applications using the xPC Target API
- The Internet browser interface

- Other utilities like `xpctargetspy`

The only rapid-prototyping feature available is signal logging, because the acquisition of signal data runs independently from the host, and logged data is retrieved only after the execution is stopped. Nevertheless, being able to log data allows gathering good enough information about the behavior of the target application. Signal logging becomes a very important feature in polling mode.

### Multirate Simulink Models Cannot Be Executed in Multitasking Mode on the Target PC

Because of the polling mode execution scheme, executing Simulink-based target applications in multitasking mode is not possible. The modeling of function-call subsystems to handle asynchronous events (interrupts) is not possible either. This can be a hard restriction, especially for multirate systems. Multirate systems can be executed in single-tasking mode, but because of its sequential execution scheme for all subsystems with different rates, the CPU will most likely overload for the given base sample time. As an important consequence, polling mode is only a feasible alternative to the interrupt mode if the model has a single rate or if it can be converted to a single-rate model. A single-rate model implies continuous states only, discrete states only, or mixed continuous and discrete states, if the continuous and discrete subsystems have the same rate. Use the **Format -> Sample time color** feature of Simulink to check for the single rate requirement. Additionally, set the **Mode** property in the **Simulation parameters / Solver** dialog box to **SingleTasking** to avoid a possible switch to multitasking mode. For more information on single-tasking mode compared to multitasking mode, see the Real-Time Workshop User's Guide.

### I/O Drivers Using Kernel Timing Information Cannot Be Used Within a Model

Some xPC Target drivers use timing information exported from the kernel in order to run properly, for example, for the detection of time-outs. Because the standard timing engine of the real-time kernel is not running during the entire target application execution in polling mode, timing information passed back to the drivers is incorrect. Drivers importing the header file `time_xpcimport.h` can therefore not be used. This is a current restriction only. In a future version of polling mode, all drivers will make use of the Pentium counter for getting timing information instead.

## Controlling the Target Application

As mentioned, there is no way to interact with the running target application in polling mode. This is especially restricting for the case of stopping the model execution before the application has reached the stop time that was defined before the execution started. Because polling mode tries to be as optimal as possible, any rapid-prototyping feature except signal logging is disabled. But because I/O driver blocks added to the model are fully functional, you can use I/O drivers to get to a minimal level of interactivity.

**Stopping a target application using polling mode** — You can use a low-latency digital input driver for the digital PCI board in your model, which reads in a single digital TTL signal. The signal is TTL low unless the model execution should be stopped, for which the signal changes to TTL high. The output port of the digital input driver block can be connected to the input port of a Stop simulation block, found in the standard Simulink block library. This stops the target application's execution, depending on the state of the digital input signal. You can either use a hardware switch connected to the board-specific input pin or you can generate the signal by other means. For example, you could use another digital I/O board in the host machine and connect the two boards (one in the host, the other in the target) over a couple of wires. You could then use MathWorks Data Acquisition Toolbox to drive the corresponding TTL output pin of the host board to stop the target application execution from within MATLAB.

Generally the same software/hardware setup can be used for passing other information back and forth during run time of the target application. It is important to understand that any additional feature beside signal logging has to be implemented at the model level, and it is therefore the user's responsibility to optimize for the minimal additional latency the feature introduces. For example, being able to interactively stop the target application execution has to be paid for by the introduction of an additional 1 us latency necessary to read the digital signal over the digital I/O board. But perhaps you need to read digital inputs from the plant hardware anyway and not all lines are used. In this case you get the feature for free.

## Polling Mode Performance

This is preliminary information. All benchmarks have been executed using a 1 GHz AMD Athlon machine, which is the same machine that is at the top of the list displayed by `xpcbench`.

The minimum achievable base sample time for model “minimal” (see “help xpcbench”) is 1 us with signal logging disabled and 2 us with signal logging enabled.

The minimum achievable base sample time for model “f14” (see “help xpcbench”) using an `ode4` fixed-step integration algorithm is 4 us with signal logging disabled and 5 us with signal logging enabled.

A more realistic model, which has been benchmarked, is a second-order continuous controller accessing real hardware over two16 bit A/D channels and two 16 bit D/A channels. The analog I/O board used is the fast and low-latency PMC-ADADIO from `http://www.generalstandards.com`, which is used in conjunction with some recently developed and heavily optimized (lowest latency) xPC Target drivers for this particular board. The minimum achievable base sample time for this model using an `ode4` fixed-step integration algorithm is 11 us with signal logging disabled and 12 us with signal logging enabled. This equals a sample rate of almost 100 kHz. The achievable sample time for the same model in interrupt mode is ~28 us or a sample rate of ~33 kHz. For this application, the overall performance increase using polling mode is almost a factor of 3!

# Target PC Command-Line Interface

You can interact with the xPC Target environment through the target PC command window. This interface is useful with stand-alone applications that are not connected to the host PC. This section includes the following topics:

- Using Methods and Properties on the Target PC
- Target Object Methods
- Target Object Properties
- Scope Object Methods
- Scope Object Properties
- Using Variables on the Target PC
- Variable Commands

## Using Methods and Properties on the Target PC

xPC Target uses an object-oriented environment on the host PC with methods and properties. While the target PC does not use the same objects, many of the methods on the host PC have equivalent target PC commands. The target PC commands are case sensitive, but the arguments are not case sensitive.

After you have created and downloaded a target application to the target PC, you can use the target PC commands to run and test your application.

1. On the target PC, press C or move the mouse over the command window.

   The target PC command window is activated, and a command line opens. If the command window is already activated, you do not have to press C. In this case, pressing C is taken as the first letter in a command.

2. In the **Cmd** box, type a target PC command. For example, to start your target application, type

   ```
   start <enter>
   ```

Once the command window is active, you do not have to reactivate it before typing the next command. For a list of target PC commands, see “Target Object Methods” on page 1-32, “Target Object Properties” on page 1-32, “Scope Object Methods” on page 1-34, and “Scope Object Properties” on page 1-35.

## Target Object Methods

When you are using the target PC command-line interface, target object methods are limited to starting and stopping the target application.

The following table lists the syntax for the target commands that you can use on the target PC. The equivalent MATLAB syntax is shown in the right column, and the target object name `tg` is used as an example for the MATLAB methods.

| Target PC | MATLAB |
|---|---|
| `start` | `tg.start` or `+tg` |
| `stop` | `tg.stop` or `-tg` |
| `reboot` | `tg.reboot` |

## Target Object Properties

When you are using the target PC command-line interface, target object properties are limited to parameters, signals, stop time, and sample time. Notice the difference between a parameter index (0, 1, . . .) and a parameter name (P0, P1, . . .).

The following table lists the syntax for the target commands that you can use on the target PC. The MATLAB equivalent syntax is shown in the right

column, and the target object name `tg` is used as an example for the MATLAB methods.

| Target PC Command | Description and Syntax | MATLAB Equivalent |
|---|---|---|
| `getpar` | Display the value of a block parameter using the parameter index.<br><br>Syntax: `getpar` *`parameter_index`* | `set(tg, 'parameter_name', number)` |
| `setpar` | Change the value of a block parameter using the parameter index.<br><br>Syntax: `setpar` *`parameter_index = floating_point_number`* | `get(tg, 'parameter_name')` |
| `stoptime` | Enter a new stop time. Use `inf` to run the target application until you manually stop it or reset the target PC.<br><br>Syntax: `stoptime =` *`floating_point_number`* | `tg.stoptime = number` |
| `sampletime` | Enter a new sample time.<br><br>Syntax: `sampletime =` *`floating_point_number`* | `tg.sampletime = number`<br><br>`set(tg, 'SampleTime', number)` |

| Target PC Command | Description and Syntax | MATLAB Equivalent |
|---|---|---|
| `P#` | Display or change the value of a block parameter. For example, `P2` or `P2=1.23e-4`<br><br>Syntax: *`parameter_name`*, or *`parameter_name = floating_point_number`*<br><br>*`parameter_name`* is `P0, P1, . . .` | `tg.`*`parameter_name`*<br>`tg.`*`parameter_name = floating_point_number`* |
| `S#` | Display the value of a signal. For example, S2.<br><br>Syntax: *`signal_name`*<br><br>*`signal_name`* is `S0, S1, . . .` | `tg.S#` |

## Scope Object Methods

When using the target PC command-line interface, you use scope object methods to start a scope, and add signal traces. Notice that the methods `addscope` and `remscope` are target object methods on the host PC, and notice the difference between a signal index (0, 1, . . .) and a signal name (S0, S1, . . .).

The following table lists the syntax for the target commands that you can use on the target PC. The MATLAB equivalent syntax is shown in the right column. The target object name `tg` and the scope object name `sc` are used as an example for the MATLAB methods.

| Target PC Command | Description and Syntax | MATLAB Equivalent |
|---|---|---|
| `addscope` | `addscope` *`scope_index`*<br>`addscope` | `tg.addscope(`*`scope_index`*`)`<br>`tg.addscope` |
| `remscope` | `remscope` *`scope_index`*<br>`remscope all` | `tg.remscope(`*`scope_index`*`)`<br>`tg.remscope` |

| Target PC Command | Description and Syntax | MATLAB Equivalent |
|---|---|---|
| `startscope` | `startscope` *`scope_index`* | `sc.start` or `+sc` |
| `stopscope` | `stopscope` *`scope_index`* | `sc.stop` or `-sc` |
| `addsignal` | `addsignal` *`scope_index`* `=` *`signal_index1`*`,` *`signal_index2`*`, . . .` | `sc.addsignal(`*`signal_index_vector`*`)` |
| `remsignal` | `remsignal` *`scope_index`* `=` *`signal_index1`*`,` *`signal_index2`*`, . . .` | `sc.remsignal(`*`signal_index_vector`*`)` |
| `viewmode` | Zoom in to one scope, or zoom out to all scopes.<br>Syntax: `viewmode` *`scope_index`* or left-click the scope window<br>`viewmode 'all'` or right-click any scope window<br>Press function key for scope, and then press V to toggle `viewmode` | |
| `ylimit` | `ylimit` *`scope_index`*<br>`ylimit` *`scope_index`* = auto<br>`ylimit` *`scope_index`* = *`num1`*, *`num2`* | |
| `grid` | `grid` *`scope_index`* `on`<br>`grid` *`scope_index`* `off` | |

## Scope Object Properties

When using the target PC command-line interface, scope object properties are limited to those shown in the following table. Notice the difference between a scope index (0, 1, . . .) and the MATLAB variable name for the scope object on

the host PC. The scope index is indicated in the top left corner of a scope window (SC0, SC1, . . .).

If a scope is running, you need to stop the scope before you can change a scope property.

The following table lists the syntax for the target commands that you can use on the target PC. The equivalent MATLAB syntax is shown in the right column, and the scope object name `sc` is used as an example for the MATLAB methods.

| Target PC | MATLAB |
|---|---|
| `numsamples` *`scope_index`* `=` *`number`* | `sc.NumSamples =` *`number`* |
| `decimation` *`scope_index`* `=` *`number`* | `sc.Decimation =` *`number`* |
| `scopemode` *`scope_index`* `= 0 or numerical, 1 or redraw, 2 or sliding, 3 or rolling` | `sc.Mode = 'numerical', 'redraw', 'sliding', 'rolling'` |
| `triggermode` *`scope_index`* `= 0, freerun, 1 software, 2, signal, 3, scope` | `sc.TriggerMode = 'freerun', 'software', 'signal', 'scope'` |
| `numprepostsamples` *`scope_index`* `=` *`number`* | `sc.NumPrePostSamples =` *`number`* |
| `triggersignal` *`scope_index`* `=` *`signal_index`* | `sc.TriggerSignal =` *`signal_index`* |
| `triggersample` *`scope_index`* `=` *`number`* | `sc.TriggerSample = number` |
| `triggerlevel` *`scope_index`* `=` *`number`* | `sc.TriggerLevel =` *`number`* |
| `triggerslope` *`scope_index`* `= 0, either, 1, rising, 2, falling` | `sc.TriggerSlope = 'Either', 'Rising', 'Falling'` |
| `triggerscope` *`scope_index2`* `=` *`scope_index1`* | `sc.TriggerScope =` *`scope_index1`* |

| Target PC | MATLAB |
|---|---|
| `triggerscopesample scope_index= integer` | `sc.TriggerSample = integer` |
| Press function key for scope, and then press S or move mouse into the scope window. | `sc.trigger` |

## Using Variables on the Target PC

Use variables to tag unfamiliar commands, parameter indices, and signal indexes with more descriptive names.

After you have created and downloaded a target application to the target PC, you can create target PC variables.

1. On the target PC, press C.

   The target PC command window is activated, and a command line opens.

2. In the **Cmd** box, type a variable command. For example, if you have a parameter that controls a motor, you could create the variables `on` and `off` by typing

   ```
   setvar on = p7 = 1
   setvar off = p7 = 0
   ```

3. Type a variable name. For example, to turn the motor on, type

   ```
   on
   ```

   The parameter P7 is changed to `1`, and the motor turns on.

## Variable Commands

The following table lists the syntax for the target commands that you can use on the target PC. The MATLAB equivalent syntax is shown in the right column.

| Target PC Command | Description and Syntax | MATLAB Equivalent |
|---|---|---|
| `setvar` | Set a variable to a value. Later you can use that variable to do a macro expansion.<br><br>Syntax: `setvar` *`variable_name = target_pc_command`*<br><br>For example, you can type `setvar aa=startscope 2`, `setvar bb=stopscope 2` | none |
| `getvar` | Display the value of a variable.<br><br>Syntax: `getvar` *`variable_name`* | none |
| `delvar` | Delete a variable.<br><br>Syntax: `delvar` *`variable_name`* | none |
| `delallvar` | Delete all variables.<br><br>Syntax: `delallvar` | none |
| `showvar` | Display a list of variables.<br><br>Syntax: `showvar` | none |

# Web Browser Interface

xPC Target has a Web server built into the kernel that allows you to interact with your target application using a Web browser. If the target PC is connected to a network, you can use a Web browser to interact with the target application from any host PC connected to the network.

Currently Microsoft Internet Explorer (Version 4.0 or later) and Netscape Navigator (Version 4.5 or later) are the only supported browsers.

This section includes the following topics:

- Connecting the Web Interface Through TCP/IP
- Connecting the Web Interface Through RS-232
- Using the Main Page
- Changing WWW Properties
- Viewing Signals with a Web Browser
- Viewing Parameters with a Web Browser
- Changing Access Levels to the Web Browser

## Connecting the Web Interface Through TCP/IP

If your host PC and target PC are connected with a network cable, you can connect the target application on the target PC to a Web browser on the host PC.

The TCP/IP stack on the xPC Target kernel supports only one simultaneous connection, because its main objective is real-time applications. This connection is shared between MATLAB and the Web browser. This also means that only one browser or MATLAB is able to connect at one time.

Before you connect your Web browser on the host PC, you must load a target application onto the target PC. The target application does not have to be running, but it must be loaded. Also, your browser must have JavaScript and StyleSheets turned on.

1 In the MATLAB window, type

```
xpcwwwenable
```

MATLAB is disconnected from the target PC, and the connection is reset for connecting to another client. If you do not use this command, your Web browser might not be able to connect to the target PC.

**2** Open a Web browser. In the address box, enter the IP address and port number you entered in the **xPC Target Setup** window. For example, if the target computer IP address is 192.168.0.1 and the port is 22222, type

```
http://192.168.0.1:22222/
```

The browser loads the xPC Target Web interface frame and pages.

## Connecting the Web Interface Through RS-232

If the host PC and target PC are connected with a serial cable instead of a network cable, you can still connect the target application on the target PC to a Web browser on the host PC. xPC Target includes a TCP/IP to RS-232 mapping application. This application runs on the host PC and writes whatever it receives from the RS-232 connection to a TCP/IP port, and it writes whatever is receives from the TCP/IP port to the RS-232 connection. TCP/IP port numbers must be less than $2^{16} = 65536$.

Before you connect your Web browser on the host PC, you must load a target application onto the target PC. The target application does not have to be running, but it must be loaded. Also, your Web browser must have JavaScript and StyleSheets turned on.

**1** In the MATLAB window, type

`xpcwwwenable` or `close(xpc)`

MATLAB is disconnected from the target PC, leaving the target PC ready to connect to another client. The TCP/IP stack of the xPC Target kernel supports only one simultaneous connection. If you do not use this command, the TCP/IP to RS-232 gateway might not be able to connect to the target PC.

**2** Open a DOS command window, and enter the command to start the TCP/IP to RS-232 gateway. For example, if the target PC is connected to COM1 and you would like to use the TCP/IP port `22222`, type the following:

```
c:\MATLABR12\toolbox\rtw\targets\xpc\xpc\bin\xpctcp2ser -v -t
22222 -c 1
```

The TCP/IP to RS-232 gateway starts running, and the DOS command window displays the message

```
*--------------------------------------------------------------*
*             xPC Target TCP/IP to RS-232 gateway               *
*               Copyright 2002 The MathWorks                   *
*--------------------------------------------------------------*
Connecting COM to TCP port 22222
Waiting to connect
```

If you did not close the MATLAB to target application connection, then `xpxtcp2ser` displays the message `Could not initialize COM port`.

**3** Open a Web browser. In the address box, enter

```
http://localhost:2222/
```

The Web browser loads the xPC Target Web interface pages.

**4** Using the Web interface, start and stop the target application, add scopes, add signals, and change parameters.

**5** In the DOS command window, press Ctrl+C.

The TCP/IP to RS-232 Gateway stops running, and the DOS command window displays the message

```
interrupt received, shutting down
```

The gateway application has a handler that responds to Ctrl-C by disconnecting and shutting down cleanly. In this case, Ctrl-C is not used to abort the application.

**6** In the MATLAB Command Window, type

```
xpc
```

MATLAB reconnects to the target application and lists the properties of the target object.

If you did not close the gateway application, MATLAB displays the message

```
Error in ==>
C:\MATLABR13\toolbox\rtw\targets\xpc\xpc\@xpc\xpc.m
On line 31 ==> sync(xpcObj);
```

To correct this problem, you must close MATLAB and then restart it.

## Syntax for the xpctcp2ser Command

The syntax for the `xpctcp2ser` command is

```
xpctcp2ser [-v] [-n] [-t tcpPort] [-c comPort]
xpctcp2ser -h
```

The options are described in the following table.

| Command-Line Option | Description |
|---|---|
| `-v` | Verbose mode. Produces a line of output every time a client connects or disconnects. |
| `-n` | Allows nonlocal connections. By default, only clients from the same computer that the gateway is running on are allowed to connect. This option allows anybody to connect to the gateway.<br><br>If you do not use this option, only the host PC that is connected to the target PC with a serial cable will be able to connect to the selected port. For example, if you start the gateway on your host PC, with the default ports, you can type in the Web browser `http://localhost:2222` . However, if you try to connect to `http://Domainname.com:22222`, you will probably get a connection error. |

| Command-Line Option | Description |
|---|---|
| `-t tcpPort` | Use TCP port *tcpPort*. Default `t` is 22222. For example, to connect to port 20010, type `-t 20010`. |
| `-h` | Print a help message. |
| `-c comPort` | Use COM port *comPort* (1 <= *comPort* <= 4). Default is 1. For example, to use COM2, type `-c 2`. |

## Using the Main Page

The **Main** page is divided into four parts, one below the other. The four parts are **System Status**, **xPC Target Properties**, **Navigation**, and **WWW Properties**.

After you connect a Web browser to the target PC, you can use the **Main** page to control the target application.

1 In the left frame, click the **Refresh** button.

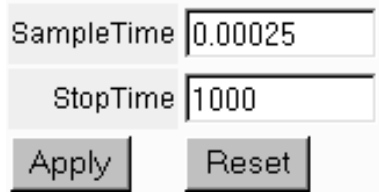
   System status information in the top cell is uploaded from the target PC. If the right frame is either the **Signals List** page or the **Screen Shot** page, updating the left frame also updates the right frame.


2 Click the **Start Execution** button.

The target application begins running on the target PC, the **Status** line is changed from **Stopped** to **Running**, and the **Start Execution** button text changes to **Stop Execution**.

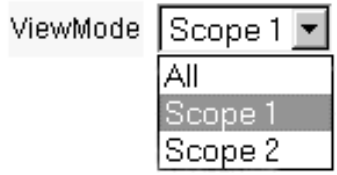
3 Update the execution time and average task execution time (TET). Click the **Refresh** button. To stop the target application, click the **Stop Execution** button.

4 Enter new values in the **StopTime** and **SampleTime** boxes, then click the **Apply** button. You can enter `-1` or `Inf` in the **StopTime** box for an infinite stop time.




The new property values are downloaded to the target application. Notice that the **SampleTime** box is visible only when the target application is stopped. You cannot change the sample time while a target application is running.

5 Select scopes to view on the target PC. From the **ViewMode** list, select one or all of the scopes to view.




**Note** The **ViewMode** control is visible only if you add two or more scopes to the target PC.

## Changing WWW Properties

The **WWW Properties** cell in the left frame contains fields that affect the display on the Web interface itself, and not the application. There are two fields: maximum signal width to display, and refresh interval.

1 In the **Maximum Signal Width** box enter `-1`, `Inf` (all signals), `1` (show only scalar signals), `2` (show scalar and vector signals less than or equal to 2 wide), or `n` (show signals with a width less than or equal to `n`).

Signals with a width greater than the value you enter are not displayed on the **Signals** page.

2 In the **Refresh Interval** box, enter a value greater than 10. For example, enter 20.

   The signal page updates automatically every `20` seconds. Entering `-1` or `Inf` does not automatically refresh the page.

Sometimes, both the frames try to update simultaneously, or the auto refresh starts before the previous load has finished. This problem can happen with slow network connections. In this case, increase the refresh interval or manually refresh the browser (set the **Refresh Interval** = `Inf`).

This can also happen when you are trying to update a parameter or property at the same time as the page is automatically refreshing.

Sometimes, when a race condition occurs, the browser becomes confused about the format, and you might have to refresh it. This should not happen often.

## Viewing Signals with a Web Browser

The **Signals** page is a list of the signals in your model.

After you connect a Web browser to the target PC you can use the **Signals** page to view signal data.

1 In the left frame, click the **Signals** button.

   The **Signals** page is loaded in the right frame with a list of signals and the current values.

2 On the **Signals** page in the right frame, click the **Refresh** button.

   The **Signals** page is updated with the current values. Vector/matrix signals are expanded and indexed in the same column-major format that MATLAB uses. This can be affected by the **Maximum Signal Width** value you enter in the left frame.

3 In the left frame, click the **Screen Shot** button.

   The **Screen Shot** page is loaded and a copy of the current target PC screen is displayed. The screen shot uses the Portable Network Graphics file format `PNG`.

## Viewing Parameters with a Web Browser

The **Parameters** page displays a list of all the tunable parameters in your model. Row and column indices for vector/matrix parameters are also shown.

After you connect a Web browser to the target PC, you can use the **Parameters** page to change parameters in your target application while it is running in real time.

1 In the left frame, click the **Parameters** button.

   The **Parameter List** page is loaded into the right frame.

   If the parameter is a scalar parameter, the current parameter value is shown in a box that you can edit.

   If the parameter is a vector/matrix, there is a button that takes you to another page that displays the vector/matrix (in the correct shape) and enables you to edit the parameter.

2 In the **Value** box, enter a new parameter value, and then click the **Apply** button.

## Changing Access Levels to the Web Browser

The Web browser interface allows you to set access levels to the target application. The different levels limit access to the target application. The highest level, 0, is the default level and allows full access. The lowest level, 4, only allows signal monitoring and tracing with your target application.

1 In the Simulink window, click **Simulation Parameters**. On the **Simulation Parameters** dialog box, click the **Real-Time Workshop** tab.

   Access levels are set in the **System target file** box. For example, to set the access level to `1`, enter

   ```
   xpctarget.tlc -axpcWWWAccessLevel=1
   ```

   The effect of not specifying -axpcWWWAccessLevel is that the highest access level (0) is set.

2 Click **OK**.

The various fields disappear, depending on the access level. For example, if your access level does not allow you access to the parameters, you do not see the button for parameters.

There are various access levels for monitoring, which allow different levels of hiding. The proposed setup is described below. Each level builds on the previous one, so only the incremental hiding of each successive level is described.

**Level 0** — Full access to all pages and functions.

**Level 1** — Cannot change the sample and stop times. Cannot change parameters, but can view parameters.

**Level 2** — Cannot start and stop execution of the target application or log data.

**Level 3** — Cannot view parameters. Cannot add new scopes, but can edit existing scopes.

**Level 4** — Cannot edit existing scopes on the **Scopes** page. Cannot add or remove signals on the **Scopes** page. Cannot view the **Signals** page and the **Parameters** page, and cannot get scope data.

2

# Graphical User Interfaces

You can run and test your target application using the MATLAB command-line interface or the Simulink block diagram for your application. You cannot modify these interfaces, but you can use special blocks provided with xPC Target to interface signals and parameters from a target application to a custom GUI application. This chapter includes the following section:

# xPC Target Interface Blocks to Simulink Models

You can use Simulink to create a custom graphical user interface (GUI) for your xPC Target application. You do this by creating an user interface model with Simulink and add-on products like the Virtual Reality Blockset and Altia Design (a third-party product). This section includes the following topics:

- **Simulink User Interface Model** — Simulink model with xPC Target interface blocks to your target application and interface blocks to graphical elements and interfaces.
- **Creating a Custom Graphical Interface** — The process for creating a custom graphical interface includes tagging parameters and signals, and then creating a Simulink user interface model with interface blocks to these parameters and signals.
- **To xPC Target Block** — Simulink blocks that take new parameter values from graphical elements and download those values to your target application.
- **From xPC Target Block** — Simulink blocks that upload signal data from your target application and pass that data to graphical elements for visualization.

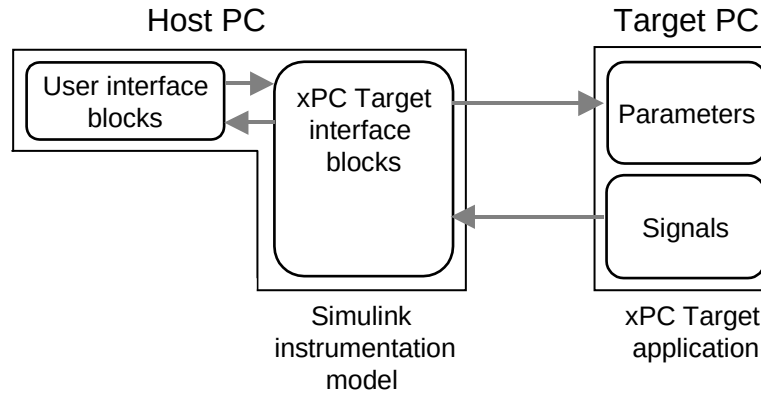
## Simulink User Interface Model

An user interface model is a Simulink model containing Simulink blocks from add-on products and interface blocks from xPC Target. This user interface model can connect to a custom graphical interface using Virtual Reality Toolbox or Altia products. The user interface model runs on the host PC and communicates with your target application running on the target PC using To xPC Target and From xPC Target blocks.

The graphical interface allows you to change parameters by downloading them to the target PC, and to visualize signals by uploading data to the host PC.

**Virtual Reality Toolbox** — The Virtual Reality Toolbox provides Simulink blocks that communicate with xPC Target interface blocks. These blocks then communicate to a graphical interface. This graphical interface is a VRML world displayed with a Web browser using a VRML plug-in.

**Altia Design** — Altia also provides Simulink blocks that communicate with xPC Target interface blocks. These blocks then communicate with Altia's graphical interface or with a Web browser using the Altia ProtoPlay plug-in.




## Creating a Custom Graphical Interface

xPC Target provides Simulink interface blocks to connect graphical interface elements to your target application. The steps for creating your own custom user interface are listed below.

1. In the Simulink target application model, decide which block parameters and block signals you want to have access to through graphical interface control devices and graphical interface display devices.

2. Tag all block parameters in the Simulink model that you want to be connected to a control device. See "Marking Block Parameters" on page 2-7.

3. Tag all signals in Simulink model that you want to be connected to a display device. See "Marking Block Signals" on page 2-10.

4. In MATLAB, run the function `xpcsliface('model_name')` to create the user interface template model. This function generates a new Simulink model containing only the xPC Target interface blocks (To xPC Target and From xPC Target) defined by the tagged block parameters and block signals in the target application model.

5 To the user interface template model, add Simulink interface blocks from add-on products (Virtual Reality Toolbox, Altia Design).

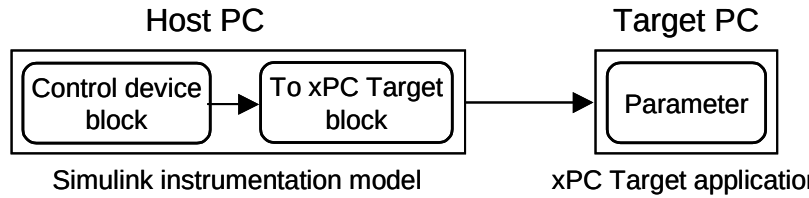
- You can connect Altia blocks to the xPC Target To PC Target interface blocks. To xPC Target blocks on the left should be connected to control devices.
- You can connect Altia and Virtual Reality Toolbox blocks to the xPC Target From PC Target interface blocks. From xPC Target blocks on the right should be connected to the display devices.

You can position these blocks to your liking.

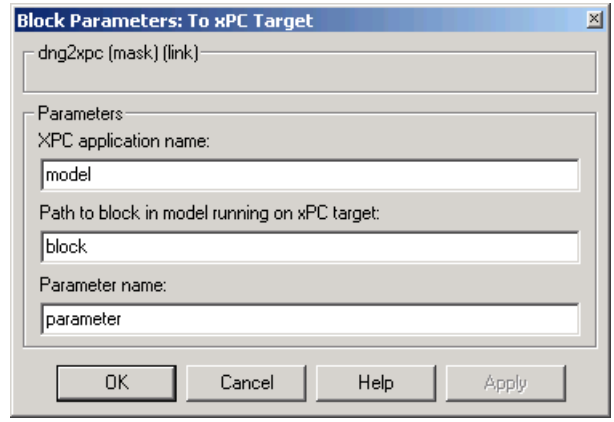
6 Start both the xPC target application and the Simulink user interface model that represents the xPC Target application.

## To xPC Target Block

This block behaves as a sink and usually receives its input data from a control device. The purpose of this block is to write a new value to a specific parameter on the target application.




This block is implemented as an M-file S-function. The block is optimized so that it only changes a parameter on the target application when the input value differs from the value that existed at the last time step. This block uses the parameter downloading feature of the xPC command-line interface.




### Block Parameters

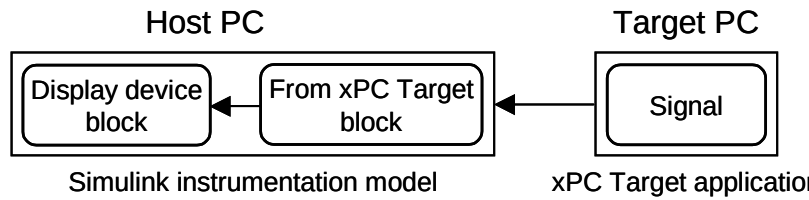
**xPC Target application name** — The function `xpcsliface` automatically enters this name. It is the same name as the Simulink model that xPC Target uses to build the target application.

**Path to block in model running on xPC target** — The function `xpcsliface` automatically enters this name and uses it to access the block identifier.

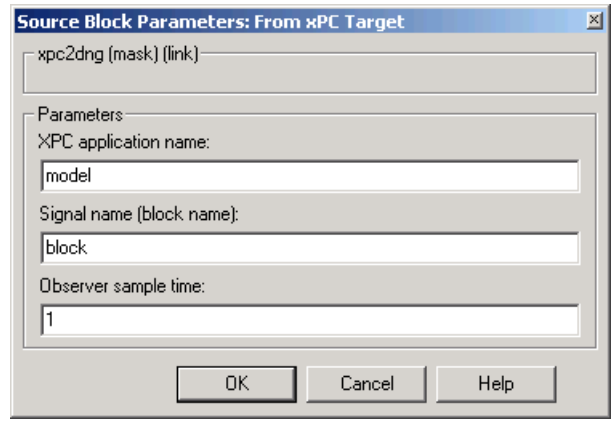
**Parameter name** — The function `xpcsliface` automatically determines this name and enters it. Note that the parameter name might not match the label name for that parameter in the **Block Parameters** dialog box. For example, the label name for a gain block is `Constant value`, but the parameter name is **Value**.

## From xPC Target Block

This block behaves like a source and its output is usually connected to the input of a display device.




Because only one numerical value per signal is uploaded during a time step, the number of samples of a scope object is set to 1. The block uses the signal tracing capability of the xPC Target command-line interface and is implemented as an M-file S-function.



### Block Parameters

**xPC Target application name** — The function `xpcsliface` automatically enters this name. It is the same name as the Simulink model that xPC Target uses to build the target application.

**Signal name (block name)** — The function `xpcsliface` automatically enters this name.

**Observer sample time** — The function `xpcsliface` automatically enters the sample time for the Simulink block with this signal. It can be equal to the model base sample time or a multiple of the base sample time.

## Creating a Target Application Model

A target application model is a Simulink model that describes your physical system, a controller, and its behavior. You use this model to create a real-time target application, and you use this model to select the parameters and signals you want to connect to a custom graphical interface.

Creating a target application model is the first step you need to do before you can tag block parameters and block signals for creating a custom graphical interface.

See “Marking Block Parameters” on page 2-7 and “Marking Block Signals” on page 2-10 for descriptions of how to mark block properties and block signals.

## Marking Block Parameters

Tagging parameters in your Simulink model allows the function `xpcsliface` to create To xPC Target interface blocks. These interface blocks contain the parameters you connect to control devices in your user interface model.

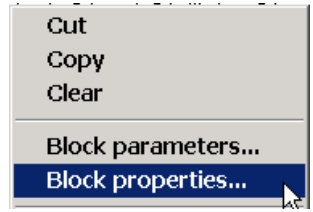
After you create a Simulink model, you can mark the block parameters. This procedure uses the model `xpctank.mdl` as an example.

1. Open a Simulink model. For example, in the MATLAB Command Window, type

   `xpc_tank` or `xpctank`

2. Point to a Simulink block, and then right-click.

3. From the menu, click **Block Properties**. Do not click **Block Parameters**.




A Block properties dialog box opens.

4 In the **Description** box, enter a tag to the parameters for this block.

For example, the SetPoint block is a constant with a single parameter that selects the level of water in the tank. Enter the tag shown below.


The tag has the following format syntax

```
xPCTag(1, . . . index_n)= label_1 . . . label_n;
```

For single dimension ports, the following syntax is also valid:

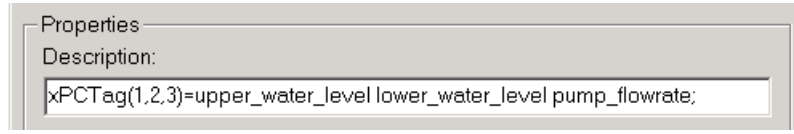
```
xPCTag=label;
```

*index_n* – Index of a block parameter. Begin numbering parameters with an index of `1`.

*label_n* – Name for a block parameter that will be connected to a To xPC Target block in the user interface model. Separate the labels with a space, not a comma.

5 Repeat steps 1 through 3 for the remaining parameters you want to tag.

For example, for the Controller block, enter the tag




For the Pump Switch and Drain Valve blocks, enter the tags



To create the To xPC blocks in an user interface model for a block with four properties, use the following syntax:

```
xPCTag(1,2,3,4)=label_1 label_2 label_3 label_4;
```

To create the To xPC blocks for the second and fourth properties in a block with at least four properties, use the following syntax:

```
xPCTag(2,4)=label_1 label_2;
```

**6** From the **File** menu, click **Save as**. Enter a filename for your model. For example, enter

```
xpc_tank1
```

You next task is to mark block signals if you have not already done so, and then create the user interface template model. See “Marking Block Signals” on page 2-10 and “Creating a Custom Graphical Interface” on page 2-3.

## Marking Block Signals

Tagging signals in your Simulink model allows the function `xpcsliface` to create From xPC Target interface blocks. These interface blocks contain the signals you connect to display devices in your user interface model.

After you create a Simulink model, you can mark the block signals. This procedure uses the model `xpc_tank1.mdl` (or `xpctank.mdl`) as an example. See "Creating a Target Application Model" on page 2-7.

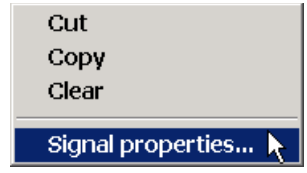
Notice that you cannot select signals on the output ports of any virtual blocks such as Subsystems and Mux blocks. Also, you cannot select signals on any Function-call, triggered signal output ports.

1. Open a Simulink model. For example, in the MATLAB Command Window, type

   `xpc_tank` or `xpc_tank1`

2. Point to a Simulink signal line, and then right-click.

3. From the menu, click **Signal Properties**.




   A **Signal Properties** dialog box opens.

4. In the **Description** box, enter a tag to the signals for this line.

   For example, the TankLevel block is an integrator with a single signal that indicates the level of water in the tank. Enter the tag shown below.


The tag has the following format syntax

```
xPCTag(1, . . . index_n)=label_1 . . . label_n;
```

For single dimension ports, the following syntax is also valid:

```
XPCTag=label:
```

| | |
|---|---|
| *index_n* | Index of a signal within a vector signal line. Begin numbering signals with an index of 1. |
| *label_n* | Name for a signal that will be connected to a From xPC Target block in the user interface model. Separate the labels with a space, not a comma. |

To create the From xPC blocks in an user interface model for a signal line with four signals (port dimension of 4), use the following syntax:

```
xPCTag(1,2,3,4)=label_1 label_2 label_3 label_4;
```

To create the From xPC blocks for the second and fourth signals in a signal line with at least four signals, use the following syntax:

```
xPCTag(2,4)=label_1 label_2;
```

5 From the **File** menu, click **Save as**. Enter a filename for your model. For example, enter

```
xpc_tank1
```

You next task is to mark block parameters if you have not already done so. See “Marking Block Parameters” on page 2-7. If you have already marked block signals, return to “Creating a Custom Graphical Interface” on page 2-3 for additional guidance on creating a user interface template model.

3

# Embedded Option

The xPC Target Embedded Option allows you to boot the target PC from a device other than a floppy disk drive, such as a hard disk or flash memory. It also allows you to create stand-alone applications on the target PC independent from the host PC. This chapter includes the following sections:

| | |
|---|---|
| Introduction (p. 3-2) | Learn about the different types of embedded target applications you can create using the xPC Target Embedded Option |
| Embedded Option Setup (p. 3-7) | Configure xPC Target to generate embedded target applications and create a DOS system boot disk |
| DOSLoader Target Applications (p. 3-11) | Create a target application that boots from a device other than a floppy disk drive |
| Stand-Alone Target Applications (p. 3-13) | Create a target application that runs on the target PC disconnected from the host PC, and optionally, boots from a device other than a floppy disk drive |

# Introduction

The xPC Target Embedded Option has two modes which create two different types of embedded target applications. This section includes the following topics:

- Overview
- DOSLoader Mode Overview
- StandAlone Mode Overview
- Software Architecture
- Restrictions

## Overview

The xPC Target Embedded Option allows you to boot the xPC Target kernel not only from a floppy disk drive, but also from other devices, including a flash disk or a hard disk drive. By using the xPC Target Embedded Option, you can configure a target PC to automatically start execution of your embedded application for continuous operation each time the system is booted. You can use this capability to deploy your own real-time applications on target PC hardware.

The xPC Target Embedded Option extends the xPC Target base product by adding two additional modes of operation:

- **DOSLoader** — Use this mode of operation to start the kernel on the target PC not only from a floppy disk, but optionally to start it from a flash disk or a hard disk. The target PC then waits for the host computer to download a real-time application either using the RS232 serial connection or using TCP/IP network communication. Control and setting of starting, stopping, parameters, tracing, and other properties can be achieved from either the host PC or from the target PC.
- **StandAlone** — This mode extends the `DOSLoader` mode. After starting the kernel on the target PC, `StandAlone` mode automatically starts execution of your target application for complete stand-alone operation. This eliminates the need for using a host computer and allows you to deploy real-time applications on PC hardware environments.

Whether you are using the xPC Target Embedded Option with the `DOSLoader` mode or the `StandAlone` mode, you initially boot your target PC with DOS from any boot device, then the kernel is started from DOS.

---

**Note** The xPC Target Embedded Option requires a boot device with DOS installed. DOS software and license are not included with xPC Target or with the xPC Target Embedded Option.

---

During setup of either the `DOSLoader` mode or `StandAlone` mode, the xPC Target Setup window allows you to create files for installation on the target boot device. One of these files is an `autoexec.bat` file. When DOS starts, it invokes the `autoexec.bat` file, which in turn starts the xPC Target kernel on the target PC.

If you do not provide a target application and an `autoexec.bat` file to invoke your target application, the xPC Target Embedded Option starts the kernel on your target PC and is ready to receive your target application whenever you build and download a new one from the host computer.

In comparison, when using xPC Target without the xPC Target Embedded Option, you can only download real-time applications to the target PC after booting from an xPC Target boot disk. Because of this, when using xPC Target without the Embedded Option, you are always required to use a target PC equipped with a floppy disk drive. However, there are several cases where your target system might not have a floppy disk drive or where the drive is removed after setting up the target system. These cases can be overcome by using the `DOSLoader` mode.

## DOSLoader Mode Overview

With the `DOSLoader` mode of operation, you first set up a boot device such as a floppy disk, flash disk, or a hard disk drive. This boot device must include DOS and modules from xPC Target Embedded Option. Once the kernel starts running, it awaits commands from the host computer and a target application that is downloaded from the host computer. The primary purpose of the `DOSLoader` mode is to allow you to boot from devices other than the floppy drive.

## StandAlone Mode Overview

With the `StandAlone` mode of operation, you create completely stand-alone applications that start execution automatically when the target PC is booted. There is no need for communication with a host computer to download the application after booting. Once the boot device has been set up with DOS, modules from xPC Target Embedded Option, and your target application, you boot the target PC. Upon booting, DOS invokes your `autoexec.bat` file, which invokes the kernel. However, in `StandAlone` mode, your target application is combined with the kernel in one binary `*.rtb` file. The final result is that your target application starts automatically each time the target PC is booted. By using xPC Target Embedded Option, you can deploy control systems, DSP applications, and other systems on PC hardware for use in production applications using PC hardware. Typically these production applications are found in systems where production quantities are low to moderate.

xPC Target Embedded Option also gives you the choice of using target scopes on the target PC. When using `StandAlone` mode, target scopes allow you to trace signals using the target PC monitor without any interaction from the host computer. Assuming that you do not want to view signals on the target PC, it is not necessary to use target scopes or a monitor on your target PC. In such a case, your system is able to operate as a *black box* without a monitor, keyboard, or mouse. Stand-alone applications are automatically set to continue running for an infinite time duration or until the target computer is turned off.

## Software Architecture

xPC Target Embedded Option creates additional files that you add to your target PC DOS boot device. With the `DOSLoader` mode, an `autoexec.bat` file is generated. This file enables DOS to automatically execute the file `xpcboot.com` once the target PC is booted. The file `autoexec.bat` includes an argument that invokes a `*.rtb` file containing the xPC Target kernel. Therefore, when the boot device invokes DOS, the `autoexec.bat` file then starts the xPC Target kernel. All these files are placed on a floppy disk when you click **BootDisk** from the `xpcsetup` GUI. Your real-time application is not copied to the boot device. You create the real-time application later by clicking **Build**.

The `StandAlone` mode operates in a similar fashion, with a few important differences. From the `xpcsetup` GUI, after choosing `StandAlone`, you only click **Update** to make your current selections active. When you later click **Build**, an `autoexec.bat` file and the `xpcboot.com` file are placed in a subdirectory that is

created within your current working directory. This directory is named *modelname_xpc_emb*. In addition, the build process creates your target application and combines it with the xPC Target kernel. This combined `*.rtb` file is also placed in the same *modename_xpc_emb* subdirectory. You copy these files onto any DOS boot device. Then, upon booting DOS, the file `xpcboot.com` is invoked with the kernel and with your target application. If you choose to use target scopes with your stand-alone application, you can do so provided appropriate xPC Target Scope blocks are added to your Simulink model and configured prior to code generation.

A small DOS executable called `xpcboot.com` is the core module of the Embedded Option. This module is used in both the `DOSLoader` mode and the `StandAlone` mode. The module `xpcboot.com` is executed from DOS. It loads and executes any xPC Target application. The first argument given to `xpcboot.com` is the name of the image file (`*.rtb`) to be executed. This image file contains the xPC Target kernel and options, such as whether you are communicating using a serial cable or TCP/IP, and the ethernet address you have assigned to the target PC.

Before starting the kernel, you must first boot the target PC under DOS. The module `xpcboot.com` is then automatically executed under DOS by `autoexec.bat`. To boot the target PC under DOS, you must first install DOS on the target PC boot device. The xPC Target Embedded Option does not have specific requirements as to the type of device you use to boot DOS. It is possible to boot from a floppy disk drive, hard disk drive, flash disk, or other device where you have installed DOS.

DOS is only needed to execute `xpcboot.com` and read the image file from the file system. After switching to the loaded kernel, and then executing the xPC Target application, DOS is discarded and is unavailable, unless you reboot the target PC without automatically invoking the xPC Target kernel. Once the xPC Target application begins execution, the target application is executed entirely in the protected mode using the 32-bit flat memory model.

## Restrictions

The following restrictions apply to the booted DOS environment when you use `xpcboot.com` to execute the target applications:

- The CPU must be executed in real mode.
- While loaded in memory, the DOS partition must not overlap the address range of a target application.

You can satisfy these restrictions by avoiding the use of additional memory managers like `emm386` or `qemm`. Also, you should avoid any utilities that attempt to load in high memory (for example, `himem.sys`). If the target PC DOS environment does not use a `config.sys` file or memory manager entries in the `autoexec.bat` file, there should not be any problems when running `xpcboot.com`.

It is also necessary for your **TargetMouse** setting to be consistent with your hardware. Some PC hardware might use an RS232 port for the mouse, while others use a PS2 mouse. If a mouse is not required in your application, you might choose to select **None** as your setting for the **TargetMouse**.

# Embedded Option Setup

This section includes the following topics:

- Updating the xPC Target Environment
- Creating a DOS System Disk

## Updating the xPC Target Environment

After the xPC Target Embedded Option software has been correctly installed, the xPC Target environment, visible through `xpcsetup` or `getxpcenv`, contains two new property choices for `DOSLoader` or `StandAlone`, in addition to the default `BootDisk` that you normally use with xPC Target.

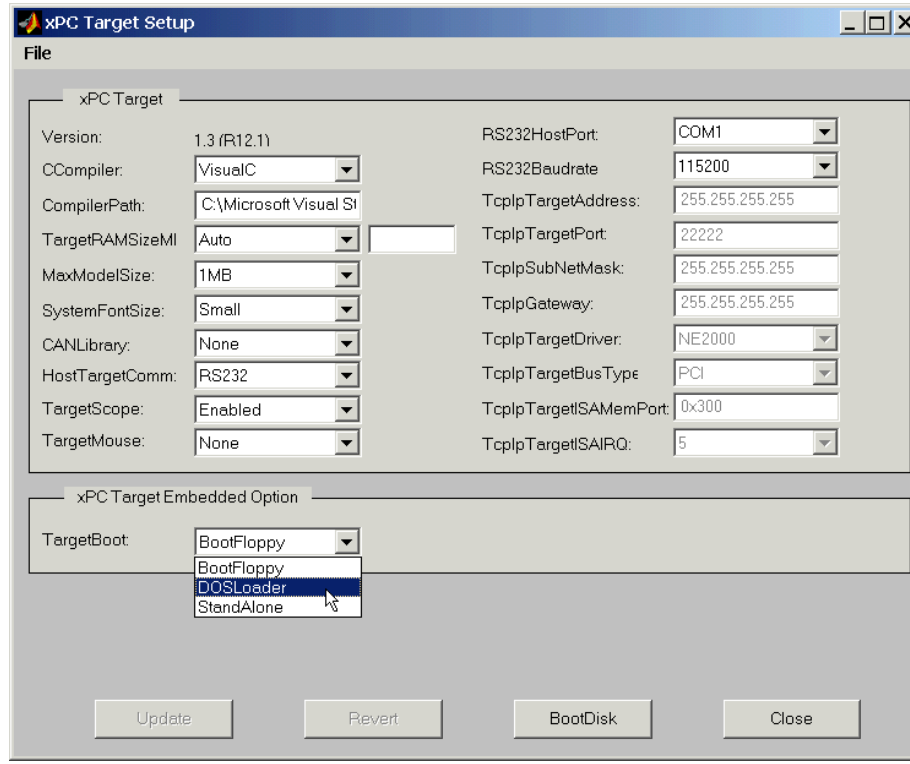
It is assumed that the xPC Target environment is already set up and working properly with the xPC Target Embedded Option disabled. If you have not already done so, we recommend you confirm this now.

You can use the function `getxpcenv` to see the current selection for **TargetBoot**, or you can view this through the xPC Target Setup window. Start MATLAB and execute the function

```
xpcsetup
```

Within the frame of the xPC Target Embedded Option, you see the property **TargetBoot,** as well as the currently selected value. The choices are

- **BootFloppy** — Standard mode of operation when xPC Target Embedded Option is not installed.
- **DOSLoader** — For invoking the kernel on the target PC from DOS.
- **StandAlone** — For invoking the kernel on the target PC from DOS and automatically starting the target application without connecting to a host computer. With this mode, the kernel and the target application are combined as a single module that is placed on the boot device.




The default setting for the property **TargetBoot** is **BootFloppy**. When using **BootFloppy**, xPC Target must first create a target boot disk, which is then used to boot the target PC.

The property **TargetBoot** can be set to two other values, namely `DOSLoader` or `StandAlone`. If the xPC Target loader is booted from any boot device with DOS installed, the value `DOSLoader` must be set as shown above. If you want to use a stand-alone application that automatically starts execution of your target application immediately after booting, specify `StandAlone`.

After changing the property value, you need to update the xPC Target environment by clicking the **Update** button in the **xPC Target Setup** window. If your choice is `DOSLoader`, you must create a new target boot disk by clicking the **BootDisk** button. Note that this overwrites the data on the inserted target boot disk as new software modules are placed on the target boot disk. If your choice is `StandAlone`, you click the **Update** button. Upon building your next real-time application, all necessary xPC Target files are saved to a subdirectory below your current working directory. This subdirectory is named with your model name with the string `'_xpc_emb'` appended, such as `xpcosc_xpc_emb`.

For more detailed information about how to use the xPC Target Setup, see Chapter 4, “Software Environment.”

## Creating a DOS System Disk

When using `DOSLoader` mode or `StandAlone` mode, you must first boot your target PC with DOS. These modes can be used from any boot device including flash disk, floppy disk drive, or a hard disk drive.

In order to boot DOS with a target boot disk, a minimal DOS system is required on the boot disk. With Windows 95, Windows 98, or DOS, you can create a DOS boot disk using the command

```
sys a:
```

---

**Note** xPC Target Embedded Option does not include a DOS license. You must obtain a valid DOS license for your target PC.

---

It is helpful to copy additional DOS-utilities to the boot disk, including

- A DOS editor to edit files
- The format program to format a hard disk or FlashRAM
- The `fdisk` program to create partitions

- The `sys` program to transfer a DOS system onto another drive, such as the hard disk drive

A `config.sys` file is not necessary. The `autoexec.bat` file should be created to boot the loader or a stand-alone xPC Target application automatically. This is described in the following sections.

# DOSLoader Target Applications

This section includes the following topics:

- Creating a Target Boot Disk for DOSLoader
- Creating a Target Application for DOSLoader

## Creating a Target Boot Disk for DOSLoader

As the first step, we assume you have created a DOS system disk and updated the xPC Target environment by setting the property **TargetBoot** to `DOSLoader`. From the **xPC Target Setup** window, click the **BootDisk** button. xPC Target copies the necessary files to the DOS disk. The files that are added to the DOS boot disk include

- `checksum.dat`
- `autoexec.bat`
- `xpcboot.com`
- `*.rtb` (where `*` is defined in the following table)

With the `DOSLoader` mode, the correct `*.rtb` file is added to the DOS disk according to the options specified in the following table.

| xPC Target Environment | HostTargetComm: RS232 | HostTargetComm: TCP/IP |
|---|---|---|
| TargetScope: Disabled | `xpcsto`*`n`*`.rtb` | `xpctto`*`n`*`.rtb` |
| TargetScope: Enabled | `xpcsgo`*`n`*`.rtb` | `xpctgo`*`n`*`.rtb` |

**Note** The numeric value of *n* corresponds to the maximum model size. This value is either 1, 4, or 16 megabytes. The default value for *n* is 1, or a 1-megabyte maximum model size.

The file `autoexec.bat` is copied to the DOS disk. This file should contain at least the following line:

```
xpcboot xxx.rtb
```

where `xxx.rtb` is the file described in the table above. We recommend that you view this `autoexec.bat` file to confirm this.

Now the target boot disk can be removed from the host and put into the target PC disk drive. Upon rebooting the target PC, DOS is booted from the target boot disk and the `autoexec.bat` file with the result in the xPC Target loader being automatically executed. From this point onwards, the CPU runs in protected mode and DOS is discarded.

You can repeat this procedure as necessary. There are no restrictions on the number of xPC Target boot floppies that you can create. However, xPC Target and the xPC Target Embedded Option do not include DOS licenses. You must purchase valid DOS licenses for your target PCs from the supplier of your choice.

If the `xpcboot` command is not placed in the `autoexec.bat`, `xpcboot.com` is not executed when the target PC is booted. Instead, the target is finished once it has booted DOS. You can then use the DOS environment to create a DOS partition on a hard disk, format it, and transfer `xpcboot.com` and `xxx.rtb` onto it. The `autoexec.bat` file can then be placed on the hard disk and edited so that it automatically boots the xPC Target loader the next time the target PC is booted. After this step the floppy disk drive can be removed from the system. The same procedure works with flash disks and other boot devices.

## Creating a Target Application for DOSLoader

After booting the target PC as described in the proceeding section, the target PC is ready to receive xPC Target applications from the host computer. Only now, these applications are received by the `DOSLoader` component of xPC Target. In every aspect, the `DOSLoader` mode allows your target PC to operate just as it normally would when running the xPC Target after booting from a standard xPC Target boot disk. When you click **Build** for your model, the target application is downloaded to the target PC using the communication protocol you specified earlier in the **xPC Target Setup** window.

# Stand-Alone Target Applications

This section includes the following topics:

- Creating a Target Application for Stand-Alone
- Creating a Target Boot Disk for StandAlone
- Using Target Scope Blocks with StandAlone

## Creating a Target Application for Stand-Alone

After you select `StandAlone` as your **TargetBoot** entry, the xPC Target environment is ready to create completely stand-alone applications using the Real-Time Workshop **Build** button.

Once the build process has finished, a message is displayed confirming that a stand-alone application has been created. With the `StandAlone` mode, the download procedure is not automated. The files necessary for creating stand-alone operation are placed in a subdirectory below your working directory. You copy these files to your DOS boot device.

After the build process is complete, files in your subdirectory include

- `model.rtb`. This image contains the xPC Target kernel and your target application.
- `autoexec.bat`. This file is automatically invoked by DOS. It then runs `xpcboot.com` and the `*.rtb` file.
- `xpcboot.com`. This file is a static file that is part of xPC Target Embedded Option.

## Creating a Target Boot Disk for StandAlone

After making a bootable DOS boot disk, the file `autoexec.bat` file must contain at least the following line

```
xpcboot model.rtb
```

where *`model`* is the name of your Simulink model.

These files should be copied to your DOS boot disk and inserted into the target drive. By rebooting the target PC, DOS is booted from the boot disk. The `autoexec.bat` file invokes the command string shown above, which starts the kernel and the real-time application. Because of the stand-alone nature of the executed `.rtb` file, the simulation of the xPC Target application starts immediately. Interaction between the host PC and target PC is no longer possible.

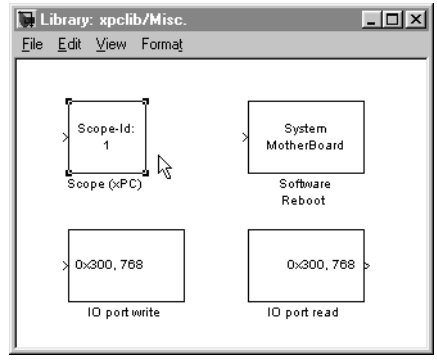
Is also possible to transfer the DOS system and stand-alone xPC Target applications to a hard disk or a flash RAM board. This offers great flexibility in creating self-starting stand-alone applications.

## Using Target Scope Blocks with StandAlone

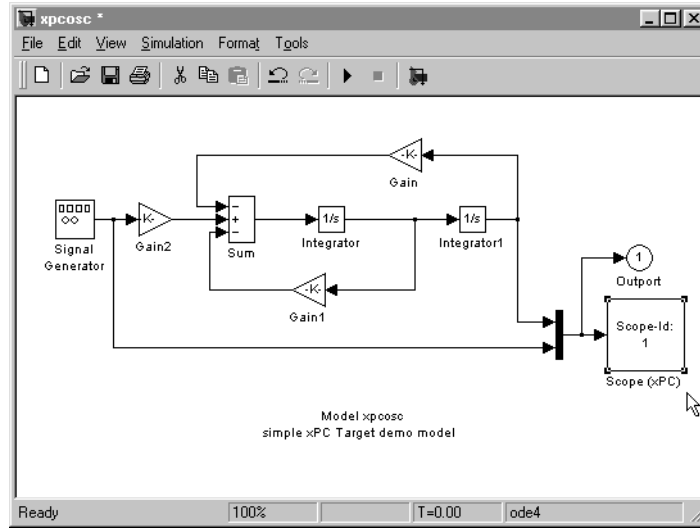
When using xPC Target Embedded Option with `StandAlone` mode, you can also use target scopes for tracing signals and displaying them on the target screen.

Because host-to-target communication is not supported with the `StandAlone` mode, scope objects of type `target` must be defined within the Simulink model before the xPC Target application is built.

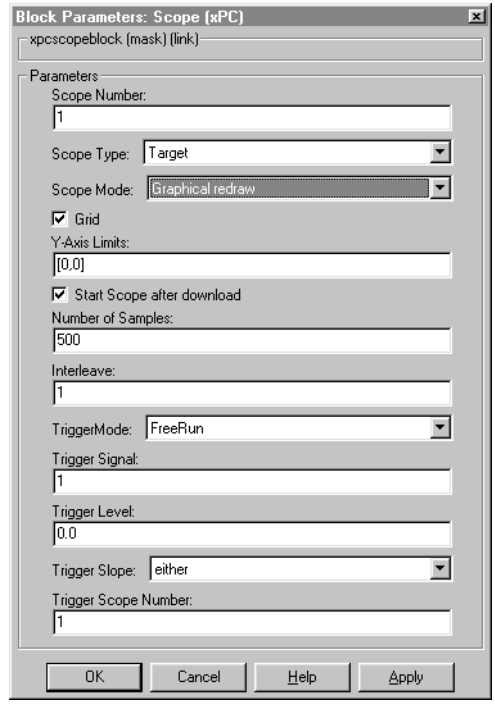
xPC Target (basic package) offers a block for such purposes.




Copy the Scope (xPC) block into your block diagram and connect the signals you would like to view to this block. Multiple signals can be used, provided a Mux block is used to bundle them.



It is necessary to edit the **Scope (xPC)** dialog box and confirm that the check box entry for **Start Scope after download** is selected, as shown in the following dialog box.




This setting is required to enable target scopes to begin operating as soon as the application starts running. The reason this setting is required is that the host PC is not available in `StandAlone` mode to issue a command that would start scopes. With these settings, click **Build** and copy the files from your *`modelname`*`_xpc_emb` subdirectory to your boot disk. Then boot your target PC. When the target application starts to run, the target scopes start automatically. A monitor is needed on your target PC to view the results.

---

**Note** When using target scopes with `StandAlone` mode, you must specify the **Scope Type** as `target` prior to generating code.

---

4

# Software Environment

The xPC Target environment defines the connections and communication between the host and target computers. It also defines the build process for a real-time application. This chapter includes the following sections:

# Environment Reference

The xPC Target environment defines the software and hardware environment of the host PC as well as the target PC. An understanding of the environment properties will help you to correctly configure the xPC Target environment. This section includes the following topics:

- **Environment Properties** — List of properties with a brief description
- **Environment Functions** — List of functions to view and change environment properties

## Environment Properties

The environment properties define communication between the host PC and target PC, the type of C compiler and its location, and the type of target boot floppy created during the setup process. You can view and change these properties using the environment functions or the Setup window.

| Environment Property | Description |
|---|---|
| `Version` | xPC Target version number. Read-only. |
| `Path` | xPC Target root directory. Read-only. |
| `CCompiler` | Values are `'Watcom'` or `'VisualC'`. From the Setup window **CCompiler** list, select either `Watcom` or `VisualC`. |
| `CompilerPath` | Value is a valid compiler root directory. Enter the path where you installed a Watcom C/C++ or Microsoft Visual C/C++ compiler.<br><br>If the path is invalid or the directory does not contain the compiler, an error message appears when you use the function `updatexpcenv` or build a target application. |

| Environment Property | Description |
|---|---|
| `TargetRAMSizeMB` | Values are `'Auto'` or '*`MB of target RAM`*'.<br><br>From the Setup window **TargetRAMSizeMB** list, select either `Auto` or `Manual`. If you select `Manual`, enter the amount of RAM, in megabytes, installed on the target PC. This property is set by default to **Auto**.<br><br>**TargetRAMSizeMB** defines the total amount of installed RAM in the target PC. This RAM is used for the, kernel, target application, data logging, and other functions that use the heap.<br><br>If **TargetRAMSizeMB** is set to **Auto**, the target application automatically determines the amount of memory up to 64 MB. If the target PC does not contain more than 64 MB of RAM, or you do not want to use more then 64 MB, select **Auto**. If the target PC has more than 64 MB of RAM, and you want to use more than 64 MB, select **Manual**, and enter the amount of RAM installed in the target PC. |
| `MaxModelSize` | Values are `'1MB'`, `'4MB'`, or `'16MB'`.<br><br>From the Setup window **MaxModelSize** list, select either `1 MB`, `4 MB`, or `16 MB`.<br><br>Choosing the maximum model size reserves the specified amount of memory on the target PC for the target application. The remaining memory is used by the kernel and by the heap for data logging.<br><br>Selecting too high a value leaves less memory for data logging. Selecting too low a value does not reserve enough memory for the target application and creates an error. |

| Environment Property | Description |
|---|---|
| `SystemFontSize` | Values are `'Small'` or `'Large'`.<br><br>From the Setup window **SystemFontSize** list, select either `Small` or `Large`.<br><br>The xPC Target GUIs use this information to change the font size. |
| `CANLibrary` | Values are `'None'`, `'200 ISA'`, `'527 ISA'`, `'1000 PCI'`, `'1000 MB PCI'`, or `'PC104'`.<br><br>From the Setup window **CANLibrary** list, select `None`, `200 ISA`, `527 ISA`, `1000 PCI`, `1000 MB PCI`, or `PC104`. |
| `HostTargetComm` | Values are `'RS232'` or `'TcpIp'`.<br><br>From the Setup window **HostTargetComm** list, select either `RS232` or `TCP/IP`.<br><br>If you select `RS232`, you also need to set the property **RS232HostPort**. If you select `TCP/IP`, then you also need to set all properties that start with **TcpIp**. |
| `RS232HostPort` | Values are `'COM1'` or `'COM2'`.<br><br>From the Setup window **RS232HostPort** list, select either `COM1` or `COM2` for the connection on the host computer. xPC Target automatically determines the COM port on the target PC.<br><br>Before you can select an RS232 port, you need to set the **HostTargetComm** property to `RS232`. |
| `RS232Baudrate` | Values are `'115200'`, `'57600'`, `'38400'`, `'19200'`, `'9600'`, `'4800'`, `'2400'`, or `'1200'`.<br><br>From the **RS232Baudrate** list, select 115200, 57600, 38400, 19200, 9600, 4800, 2400, or 1200. |

| Environment Property | Description |
|---|---|
| `TcpIpTargetAddress` | Value is `'xxx.xxx.xxx.xxx'`.<br><br>In the Setup window **TcpIpTargetAddress** box, enter a valid IP address for your target PC. Ask your system administrator for this value.<br><br>For example, `192.168.0.1` |
| `TcpIpTargetPort` | Value is `'xxxxx'`.<br><br>In the Setup window **TcpIpTargetPort** box, enter a value greater than `20000`.<br><br>This property is set by default to `22222` and should not cause any problems. The number is higher than the reserved area (telnet, ftp, ...) and it is only of use on the target PC. |
| `TcpIpSubNetMask` | Value is `'xxx.xxx.xxx.xxx'`.<br><br>In the Setup window **TcpIpSubNetMask** text box, enter the subnet mask of your LAN. Ask your system administrator for this value.<br><br>For example, your subnet mask could be `255.255.255.0`. |

| Environment Property | Description |
|---|---|
| `TcpIpGateway` | Value is '*xxx.xxx.xxx.xxx*'.<br><br>In the Setup window **TcpIpGateway** box, enter the IP address for your gateway. This property is set by default to `255.255.255.255`, which means that a gateway is not used to connect to the target PC.<br><br>If you communicate with your target PC from within a LAN that uses gateways, and your host and target computers are connected through a gateway, then you need to enter a value for this property. If your LAN does not use gateways, you do not need to change this property. Ask your system administrator. |
| `TcpIpTargetDriver` | Values are `'NE2000'` or `'SMC91C9X'`.<br><br>From the Setup window **TcpIpTargetDriver** list, select `NE2000`, `SMC91C9X`, `I82559`, or `RTLANCE`. The Ethernet card provided with xPC Target uses the `NE2000` driver. |
| `TcpIpTargetBusType` | Values are `'PCI'` or `'ISA'`.<br><br>From the Setup window **TcpIpTargetBusType** list, select either `PCI` or `ISA`. This property is set by default to `PCI`, and determines the bus type of your target PC. You do not need to define a bus type for your host PC, which can be the same or different from the bus type in your target PC.<br><br>If **TcpIpTargetBusType** is set to **PCI**, then the properties **TcpIpISAMemPort** and **TcpIpISAIRQ** have no effect on TCP/IP communication.<br><br>If you are using an ISA bus card, set **TcpIpTargetBusType** to **ISA** and enter values for **TcpIpISAMemPort** and **TcpIpISAIRQ**. |

| Environment Property | Description |
|---|---|
| `TcpIpTargetISAMemPort` | Value is `'0x`*nnnn*`'`.<br><br>If you are using an ISA bus Ethernet card, you must enter values for the properties **TcpIpISAMemPort** and **TcpIpISAIRQ**. The values of these properties must correspond to the jumper settings or ROM settings on your ISA bus Ethernet card.<br><br>On your ISA bus card, assign an IRQ and I/O port base address by moving the jumpers on the card.<br><br>Set the I/O port base address to around 0x300. If one of these hardware settings leads to a conflict in your target PC, choose another I/O port base address and make the corresponding changes to your jumper settings. |
| `TcpIpTargetISAIRQ` | Value is `'`*n*`'` where *n* is between 4 and 15.<br><br>If you are using an ISA bus Ethernet card, you must enter values for the properties **TcpIpISAMemPort** and **TcpIpISAIRQ**. The values of these properties must correspond to the jumper settings or ROM settings on the ISA-bus Ethernet card.<br><br>On your ISA bus card, assign an IRQ and I/O-port base address by moving the jumpers on the card.<br><br>We recommend setting the IRQ to 5, 10, or 11. If one of these hardware settings leads to a conflict in your target PC, choose another IRQ and make the corresponding changes to your jumper settings. |
| `EmbeddedOption` | Values are `'Disabled'` or `'Enabled'`. This property is read-only.<br><br>**Note** The xPC Target Embedded Option is enabled only if you purchase an additional license. |

| Environment Property | Description |
|---|---|
| `TargetScope` | Values are `'Disabled'` or `'Enabled'`.<br><br>From the Setup window **TargetScope** list, select either `Enabled` or `Disabled`.<br><br>The property **TargetScope** is set by default to **Enabled**. If you set **TargetScope** to **Disabled**, the target PC displays information as text.<br><br>To use all the features of the target scope, you also need to install a keyboard and mouse on the target PC. |

| Environment Property | Description |
|---|---|
| `TargetMouse` | Values are `'None'`, `'PS2'`, `'RS232 COM1'`, `'RS232 COM2'`.<br><br>From the Setup window **TargetMouse** list, select `None`, `PS2`, `RS232 COM1`, or `RS232 COM2`.<br><br>Before you can select a target mouse, you need to set the **Target Scope** property to **Enabled**.<br><br>**TargetMouse** allows you to disable or enable mouse support on the target PC:<br><br>• If you do not connect a mouse to the target PC, you need to set this property to **None**; otherwise, the target application might not behave properly.<br>• If the target PC supports PS/2 devices (keyboard and mouse) and you connect a PS/2 mouse, set this property to **PS2**.<br>• If you connect a serial RS232 mouse to the target PC, select either `RS232 COM1` or `RS232 COM2` depending on which serial port you attached the mouse to. |
| `TargetBoot` | Values are `'BootFloppy'`, `'DOSLoader'`, or `'StandAlone'`.<br><br>From the Setup window **TargetBoot** list, select `BootFloppy`, `DOSLoader`, or `StandAlone`.<br><br>If your license file does not include the license for the xPC Target Embedded Option, the **Target Boot** box is disabled, with **BootFloppy** as your only selection. With the xPC Target Embedded Option licensed and installed, you have the additional choices of **DOSLoader** and **StandAlone**. |

## Environment Functions

The environment functions allow you to change the environment properties. The functions are listed in the following table.

| Environment Functions | Description |
|---|---|
| `getxpcenv` | List environment properties in the MATLAB window or assign the list as a cell array to a MATLAB variable. |
| `setxpcenv` | First of two steps to change environment properties. See also `updatexpcenv`. |
| `updatexpcenv` | Change the current environment properties to equal the new properties entered using the function `setxpcenv`. |
| `xpcbootdisk` | Create a boot floppy disk containing the kernel according to the current environment properties. |

# Using Environment Properties and Functions

Use the xPC Target Setup window or the MATLAB command window to enter environment properties that are independent of your model. This section includes the following topics:

- Getting a List of Environment Properties
- Saving and Loading the Environment Properties
- Changing Environment Properties with a Graphical Interface
- Changing Environment Properties with a Command-Line Interface

To enter properties specific to your model and its build procedure, see "Entering the Real-Time Workshop Parameters" on page 3-26. These properties are saved with your Simulink model.

## Getting a List of Environment Properties

To use the xPC Target functions to change environment properties, you need to know the names and allowed values of these properties. Use the following procedure to get a list of the property names, their allowed values, and their current values:

1. In the MATLAB window, type

   ```
   setxpcenv
   ```

   MATLAB displays a list of xPC Target environment properties and the allowed values. For a list of the properties, see "Environment Properties" on page 4-2

2. Type

   ```
   getxpcenv
   ```

   MATLAB displays a list of xPC Target environment properties and the current values.

Alternatively, you can use the **xPC Target Setup** dialog box to view and change environment properties. In the MATLAB window, type `xpcsetup`.

## Saving and Loading the Environment Properties

This feature makes it easy and fast to switch between different xPC Target environments:

1. In the **xPC Target Setup** window, and from the **File** menu, click **Save Settings**.

   The **Save xPC Target Environment** dialog box opens.

2. Enter the name of an environment file (`*.mat`). Select a directory, and then click **Save**.

   xPC Target saves the current environment properties.

   After you have saved an xPC Target environment, you can load those property values back into xPC Target.

3. From the **File** menu, click **Load Settings**.

   The **Load xPC Target Environment** dialog box opens.

4. Select a directory with a previously saved environment file (*.mat). Select the file, and then click **Open**.

5. In the **xPC Target Setup** window, click the **Close** button.

   If you change the environment properties but do not click the **Update** button, xPC Target displays a warning.

Even if you decide to continue with the exit process, you do not lose the values you changed. However, the current environment does not reflect the changes you made in the **xPC Target Setup** window. If you reopen the **xPC Target Setup** window, the changes you made reappear, and the **Update** button is enabled.

## Changing Environment Properties with a Graphical Interface

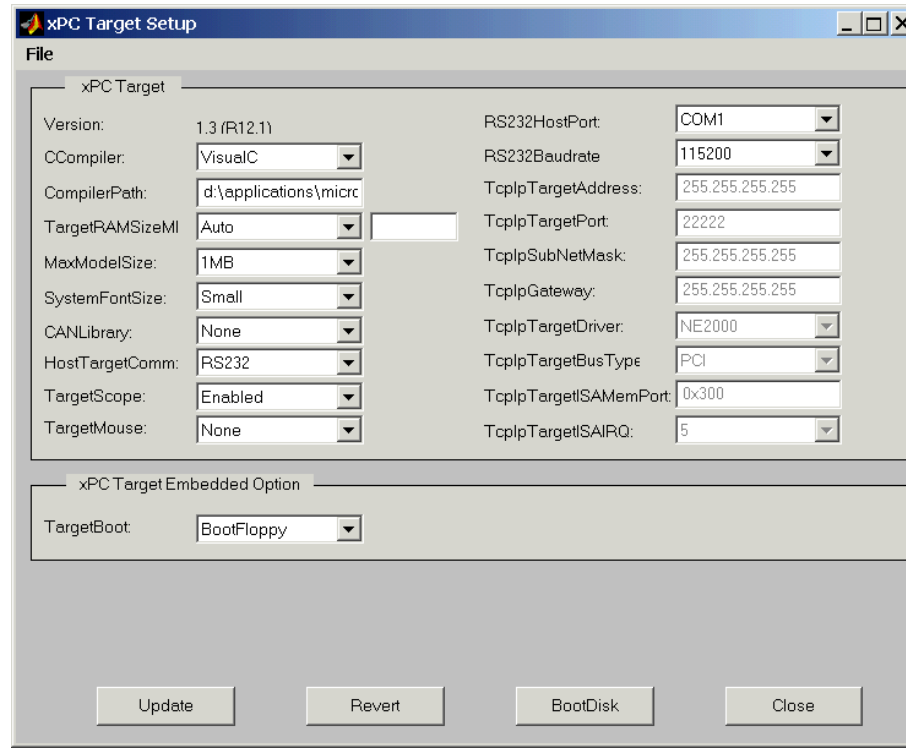
xPC Target lets you define and change environment properties. These properties include the path to the C/C++ compiler, the host PC COM port, the logging buffer size, and many others. Collectively these properties are known as the xPC Target environment.

To change an environment property using the xPC Target GUI, use the following procedure:

1 In the MATLAB Command Window, type

```
xpcsetup
```

MATLAB opens the **xPC Target Setup** window.



The **xPC Target Setup** window has two sections:

- **xPC Target**
- **xPC Target Embedded Option**

If your license does not include the xPC Target Embedded Option, the **TargetBoot** box is grayed out, with **BootFloppy** as your only selection. With the xPC Target Embedded Option, you have the additional choices of **DOSLoader** and **StandAlone**.

**2** Change properties in the environment by entering new property values in the text boxes or choosing items from the lists.

After you make changes to the environment properties, you need to update the xPC Target environment. Updating makes your changes in the **xPC Target Setup** window equal to the current property values.

**3** Click the **Update** button.

xPC Target updates the xPC Target environment and disables (grays out) the **Update** button. As long as the **Update** button is enabled, the xPC Target environment needs to be updated.

## Changing Environment Properties with a Command-Line Interface

xPC Target lets you define and change different properties. These properties include the path to the C/C++ compiler, the host COM port, the logging buffer size, and many others. Collectively these properties are known as the xPC Target environment.

You can use the command-line functions to write an M-file script that accesses the environment settings according to your own needs. For example, you could write an M-file that switches between two targets.

The following procedure shows how to change the COM port property for your host PC from COM1 to COM2:

**1** In the MATLAB window, type

```
setxpcenv('RS232HostPort','COM2')
```

The up-to-date column shows the values that you have changed, but have not updated.

```
HostTargetComm                  :RS232          up to date
RS232HostPort                   :COM1           COM2
RS232Baudrate                   :115200         up to date
```

Making changes using the function `setxpcenv` does not change the current values until you enter the update command.

**2** In the MATLAB window, type

```
updatexpcenv
```

The environment properties you changed with the function `setxpcenv` become the current values.

```
HostTargetComm                  :RS232          up to date
RS232HostPort                   :COM2           up to date
RS232Baudrate                   :115200         up to date
```

# System Functions

The system functions allow you to open xPC Target GUIs and run tests from the MATLAB window. This section includes the following topics:

- GUI Functions
- Test Functions
- xPC Target Demos

## GUI Functions

The GUI functions are listed in the following table.

| System Functions | Description |
|---|---|
| `xpcrctool` | Open the remote control tool on the host PC for running the target application on the target PC |
| `xpcscope` | Open the scope manager window on the host PC for scopes with type `host`. |
| `xpcsetup` | Open the Setup window. |
| `xpctargetspy` | Open the Target Spy window on the host PC. Use this GUI to upload the target PC screen to the host PC. |
| `xpctargetspy` | Open the Target Spy window on the host PC. Use this GUI to upload the target PC screen to the host PC. |
| `xpctgscope` | Open the scope manager window on the host PC for scopes with type `target`. |

## Test Functions

The test functions are listed in the following table.

| System Functions | Description |
|---|---|
| `getxpcpci` | Determine which PCI boards are installed in the target PC. |
| `xpctargetping` | Test the communication between the host PC and the target PC |
| `xpctest` | Test the xPC Target installation. |

## xPC Target Demos

The xPC Target demos are used to demonstrate the features of xPC Target. But they are also M-file scripts that you can view to understand how to write your own scripts for creating and testing target applications.

The following table lists the demo scripts that are provided with xPC Target.

| Demo | Filename |
|---|---|
| Parameter Sweep | `parsweepdemo` |
| Signal tracing using free-run mode | `scfreerundemo` |
| Signal tracing using software triggering | `scsoftwaredemo` |
| Signal tracing using signal triggering | `scsignaldemo` |
| Signal tracing using scope triggering | `scscopedemo` |
| Signal tracing using the target scope | `tgscopedemo` |

### To locate or edit a demo script

1 In the MATLAB window, type

```
scfreerundemo
```

MATLAB displays the location of the M-file.

```
D:\MATLAB\toolbox\rtw\targets\xpc\xpcdemos\scfreerundemo.m
```

**2** Type

```
edit scfreerundemo
```

MATLAB opens the M-file in a MATLAB editing window.

## Purpose

List environment properties assigned to a MATLAB variable

## Syntax

**MATLAB Command Line**

```
getxpcenv
```

## Description

Function for environment properties. This function displays, in the MATLAB window, the property names, the current property values, and the new property values set for the xPC Target environment.

## Examples

Return the xPC Target environment in the structure shown below. The output in the MATLAB window is suppressed. The structure contains three fields for property names, current property values, and new property values.

```
env =getxpcenv
```

```
env =
      propname: {1x25 cell}
    actpropval: {1x25 cell}
    newpropval: {1x25 cell}
```

Display a list of the environment property names, current values, and new values.

```
env =getxpcenv
```

## See Also

The xPC Target functions `setxpcenv`, `updatexpcenv`, `xpcbootdisk`, and `xpcsetup`.

# getxpcpci

## Purpose

Determine which PCI boards are installed in the target PC

## Syntax

**MATLAB Command Line**

```
getxpcpci('type_of_boards')
```

## Arguments

| | |
|---|---|
| `type_of_boards` | Values are no arguments, `'all'`, and `'supported'`. |

## Description

The information is displayed in the MATLAB window. Only devices supported by driver blocks in the xPC Target Block Library are displayed. The information includes the PCI bus number, slot number, assigned IRQ number, manufacturer name, board name, device type, manufacturer PCI ID, and the board PCI ID itself.

For a successful query:

- The host-target communication link must be working. (The function `xpctargetping` must return `success` before using the function `getxpcpci`.
- Either a target application is loaded or the loader is active. The latter is used to query for resources assigned to a specific PCI device, which have to be provided to a driver block dialog box prior to the model build process.

## Examples

Return the result of the query in the struct `pcidevs` instead of displaying it. The struct `pcidevs` is an array with one element for each detected PCI device. Each element combines the information by a set of field names. The struct contains more information compared to the displayed list, such as the assigned base addresses, the base and subclass.

```
pcidevs = getxpcpci
```

Display the supported and installed PCI devices.

```
getxpcpci('all')
```

Display the installed PCI devices, not only the devices supported by the xPC Target Block Library. This includes graphics controller, network cards, SCSI cards, and even devices that are part of the motherboard chipset (for example PCI-to-PCI bridges).

```
getxpcpci('all')
```

Display a list of the currently supported PCI devices in the xPC Target block library. The result is stored in a struct instead of displaying it.

```
getxpcpci('supported')
```

When called with the `'supported'` option, `getxpcpci` does not access the target PC.

# setxpcenv

## Purpose

Change xPC Target environment properties

## Syntax

**MATLAB Command Line**

```
setxpcenv('property_name', 'property_value')
setxpcenv('prop_name1', 'prop_val1', 'prop_name2', 'prop_val2')
setxpcenv
```

## Arguments

| | |
|---|---|
| `property_name` | Not case sensitive. Property names can be shortened as long as they can be differentiated from the other property names. |
| `property_value` | Character string. Type `setxpcenv` without arguments to get a listing of allowed values. Property values are not case sensitive. |

## Description

Function for environment properties. Enter new environment properties. If the new value is different from the current value, the property is marked as having a new value. Use the function `updatexpcenv` to change the current properties to the new properties.

The function `setxpcenv` works similarly to the function `set` of the MATLAB Handle Graphics system. The function `setxpcenv` must be called with an even number of arguments. The first argument of a pair is the property name, and the second argument is the new property value for this property.

Using the function `setxpcenv` without arguments returns a list of allowed property values in the MATLAB window.

## Examples

List the current environment properties. For a description of properties and allowed values, see "Environment Properties" on page 4-2.

```
setxpcenv
```

Change the host PC, serial communication port, to COM2.

```
setxpcenv('HostCommPort','COM2')
```

## See Also

The xPC Target functions `getxpcenv`, `updatexpcenv`, `xpcbootdisk`, and `xpcsetup`. The procedures "Changing Environment Properties with a

Graphical Interface” on page 4-12 and “Changing Environment Properties with a Command-Line Interface” on page 4-15.

# updatexpcenv

## Purpose

Change current environment properties to equal new properties

## Syntax

**MATLAB Command Line**

```
updatexpcenv
```

## Description

Function for environment properties. This procedure includes creating communication M-files as well as patching the xPC Target kernel and system DLLs. Calling the function `updatexpcenv` is necessary after new properties are entered with the function `setxpcenv`, but before creating a target boot floppy with the function `xpcbootdisk`.

## See Also

The xPC Target functions `setxpcenv`, `getxpcenv`, `updatexpcenv`, `xpcbootdisk`, and `xpcsetup`. The procedures “Changing Environment Properties with a Graphical Interface” on page 4-12 and “Changing Environment Properties with a Command-Line Interface” on page 4-15.

## Purpose

Create xPC Target boot disk, and confirm the current environment properties

## Syntax

**MATLAB Command Line**

```
xpcbootdisk
```

## Description

Function for environment properties. This function creates an xPC target boot floppy for the current xPC Target environment that has been updated with the function `updatexpcenv`. Creating an xPC Target boot floppy consists of writing the correct bootable kernel image onto the disk. You are asked to insert an empty formatted floppy disk into the floppy drive.

All existing files are erased by the function `xpcbootdisk`. If the inserted floppy disk already is an xPC Target boot disk for the current environment settings, this function exits without writing a new boot image to the floppy disk. At the end, a summary of the creation process is displayed.

If you update the environment, you need to update the target boot floppy for the new xPC environment with the function `xpcbootdisk`.

## Examples

To create a boot floppy disk, in the MATLAB window, type

```
xpcbootdisk
```

## See Also

The xPC Target functions `setxpcenv`, `getxpcenv`, `updatexpcenv`, `xpcbootdisk`, and `xpcsetup`. The procedures “System Functions” on page 4-16 and “System Functions” on page 4-16.

## Purpose

Open a the remote control tool on the host PC

## Syntax

**MATLAB Command Line**

```
xpcrttool
```

## Description

This graphical user interface (GUI) allows you to control your target application running on the target PC. It also allows you to add scopes, acquire signal data and tune parameters using the xPC Target Simulink browser.

## See Also

The xPC Target functions `xpcscope`, `xpctgscope`, and the procedure “Signal Tracing” on page 4-3.

## Purpose

Open a scope manager window on the host PC

## Syntax

**MATLAB Command Line**

```
xpcscope
```

## Description

This graphical user interface (GUI) allows you to define scopes that display on your host PC, choose signals, and control the data acquisition process.

## See Also

The xPC Target function `xpctgscope` and the procedure “Signal Tracing” on page 4-3.

# xpcsetup

## Purpose

Open the Setup window

## Syntax

**MATLAB Command Line**

```
xpcsetup
```

## Description

This graphical user interface (GUI) allows you to

- Enter and change environment properties
- Create an xPC Target boot floppy disk

## See Also

Functions — `setxpcenv`, `getxpcenv`, `updatexpcenv`, `xpcbootdisk`

Procedures — “Environment Properties for Serial Communication” on page 2-11 and “Environment Properties for Network Communication” on page 2-19

## Purpose

Test communication between the host and target computers

## Syntax

**MATLAB Command Line**

```
xpctargetping
```

## Examples

Check for communication between the host PC and target PC.

```
xpctargetping
```

## Description

Pings the target PC from the host PC and returns either '`success`' or '`failed`'. If the xPC Target kernel is loaded, running, and communication is working properly, this function returns the value '`success`'.

This function works with both RS232 and TCP/IP communication.

```
ans =
success
```

## See Also

The xPC Target procedure “Testing and Troubleshooting the Installation” on page 2-28.

# xpctargetspy

## Purpose

Open an xPC Target Spy window on the host PC

## Syntax

**MATLAB Command Line**

```
xpctargetspy
```

## Description

This graphical user interface (GUI) allows you to upload displayed data from the target PC.

The behavior of this function depends on the value for the environment property `TargetScope`.

- If `TargetScope` is enabled, a single graphics screen is uploaded. The screen is not continually updated because of a higher data volume when a target graphics card is in VGA mode.

  To update the host screen with another target screen, move the pointer into the Spy window and left-click.

- If `TargetScope` is disabled, text output is transferred once every second to the host and displayed in the window.

## Examples

To open the Target Spy window, in the MATLAB window, type

```
xpctargetspy
```

## Purpose

Test the xPC Target installation

## Syntax

**MATLAB Command Line**

```
xpctest
xpctest('noreboot')
```

## Arguments

| | |
|---|---|
| `'noreboot'` | Only one possible option. Skips the reboot test. Use this option if the target hardware does not support software rebooting. Value is `'noreboot'`. |

## Description

Series of xPC Target tests to check the correct functioning of the following xPC Target tasks:

- Initiate communication between the host and target computers.
- Reboot the target PC to reset the target environment.
- Build a target application on the host PC.
- Download a target application to the target PC.
- Check communication between the host and target computers using commands.
- Execute a target application.
- Compare the results of a simulation and the target application run.

`xpctest('noreboot')` skips test 2. Use this option if target hardware does not support software rebooting.

## Examples

If the target hardware does not support software rebooting, or to skip test 2, in the MATLAB window, type

```
xpctest('noreboot')
```

## See Also

The procedures “Testing and Troubleshooting the Installation” on page 2-28 and “Test 1, Ping Target System Standard Ping” on page 2-29.

# xpctgscope

## Purpose

Open the target scope manager window

## Syntax

**MATLAB Command Line**

```
xpctgscope
```

## Description

This graphical user interface (GUI) allows you to define scopes that display on your target PC, choose signals, and control the data acquisition process.

## See Also

The xPC Target function `xpcscope` and the procedure “Signal Tracing” on page 4-3.

## Purpose

Disconnect the target PC from the current client application

## Syntax

**MATLAB Command Line**

```
xpcwwwenable
```

## Description

Use this function to disconnect the target application from MATLAB before you connect to the Web browser. You can also use this function to connect to MATLAB after using a Web browser, or to switch to another Web browser.

5

# Target Objects

xPC Target uses a target object to represent the target kernel and your target application. It also uses a scope object to represent the data acquisition part of the kernel. Use target objects to run and control real-time applications on the target PC with scope objects to collect signal data. This chapter includes the following sections:

# Target Object Reference

xPC Target uses a target object to represent the target kernel and your target application. An understanding of the target object properties and methods will help you to control and test your application on the target PC. This section includes the following topics:

- What Is a Target Object?
- Target Object Properties
- Target Object Methods

## What Is a Target Object?

A target object on the host PC represents the interface to a target application and the kernel on the target PC. You use target objects to run and control the target application.

When you change a target object property on the host PC, information is exchanged with the target PC and the target application.

To create a target object:

- Build a target application. xPC Target creates a target object during the build process.
- Use the target object constructor function `xpc`. In the MATLAB window, type `tg = xpc`.

A target object has associated properties and methods specific to that object.

## Target Object Properties

Target object properties let you access information from your target application and control its execution. You can view and change these properties using target object methods.

The properties for a target object are listed in the following table. This table includes a description of the properties and which properties you can change directly by assigning a value.

| Property | Description | Write |
|---|---|---|
| `Connected` | Communication status between the host PC and the target PC. Values are `'Yes'` or `'No'`. | |
| `Application` | Name of the Simulink model and target application built from that model. | |
| `Mode` | Type of Real-Time Workshop code generation. Values are `'Real-Time Singletasking'`, `'Real-Time Multitasking'`, or `'Accelerate'`. The default value is `'Real-Time Singletasking'`. | |
| | **Note** Even if you select `'Real-Time Multitasking'`, the actual mode can be `'Real-Time Singletasking'`. This happens if your model contains only one or two tasks and the sample rates are equal. | |
| `Status` | Execution status of your target application. Values are `'stopped'` or `'running'`. | |
| `CPUoverload` | CPU status for overload. If the target application requires more CPU time than the sample time of the model, this value is set from `'none'` to `'detected'` and the current run is stopped. Correcting CPUoverload requires either a faster processor or a larger sample time. | |

| Property | Description | Write |
|---|---|---|
| `ExecTime` | Execution time. Time, in seconds, since your target application started running. When the target application stops, the total execution time is displayed. | |
| `SessionTime` | Time since the kernel started running on your target PC. This is also the elapsed time since you booted the target PC. Values are in seconds. | |
| `StopTime` | Time when the target application stops running. Values are in seconds. The original value is set in the **Simulink Simulation Parameters** dialog box.<br><br>When the `ExecTime` reaches the `StopTime`, the application stops running. | Yes |
| `SampleTime` | Time between samples. This value equals the step size, in seconds, for updating the model equations and posting the outputs. | Yes |
| `AvgTET` | Average task execution time. This value is an average of the measured CPU times, in seconds, to run the model equations and post outputs during each sample interval. Task execution time is nearly constant, with minor deviations due to cache, memory access, interrupt latency, and multirate model execution. | |
| `MinTET` | Minimum task execution time. Corresponds to the fastest time (smallest time measured), in seconds, to update model equations and post outputs. | |

| Property | Description | Write |
|---|---|---|
| `MaxTET` | Maximum task execution time. Corresponds to the slowest time (longest time measured), in seconds, to update model equations and post outputs. | |
| `ViewMode` | Display either all scopes or a single scope on the target PC. Values are `'all'` or a single scope index. This property is active only if the environment property **TargetScope** is set to `enabled`. | Yes |
| `TimeLog` | Storage in the MATLAB workspace for the time or t-vector logged during execution of the target application. | |
| `StateLog` | Storage in the MATLAB workspace for the state or x-vector logged during execution of the target application. | |
| `OutputLog` | Storage in the MATLAB workspace for the output or y-vector logged during execution of the target application. | |
| `TETLog` | Storage in the MATLAB workspace for a vector containing task execution times during execution of the target application.<br><br>To enable logging of the TET, you need to check the **Log Task Execution Time** box located at **Simulation Parameters** dialog box **-> Real-Time Workshop** page **-> Category:xPC Target** code generation options group. | |

| Property | Description | Write |
|---|---|---|
| `MaxLogSamples` | Maximum number of samples for each logged signal within the circular buffers for `TimeLog`, `StateLog`, `OutputLog`, and `TETLog`. `StateLog` and `OutputLog` can have one or more signals.<br><br>This value is calculated by dividing the **Signal Logging Buffer Size** by the number of logged signals. The **Signal Logging Buffer Size** box is located at **Simulation Parameters** dialog box **-> Real-Time Workshop** page **-> Category:xPC Target** code generation options group. | |
| `NumLogWraps` | The number of times the circular buffer wrapped. The buffer wraps each time the number of samples exceeds `MaxLogSamples`. | |
| `LogMode` | Controls which data points are logged.<br><br>• Time-equidistant logging. Logs a data point at every time interval. Set value to `'Normal'`.<br><br>• Value-equidistant logging. Logs a data point only when an output signal from the `OutputLog` changes by a specified value (increment). Set the value to the difference in signal values. | Yes |
| `Scopes` | List of index numbers, with one index for each scope. | |
| `NumSignals` | The number of signals from your Simulink model that are available to be viewed with a scope. | |

| Property | Description | Write |
|---|---|---|
| `ShowSignals` | Flag set to view or hide the list of signals from your Simulink blocks. This list is shown when you display the properties for a target object. Values are `'on'` or `'off'`. | Yes |
| `Signals` | List of viewable signals. This list is visible only when `ShowSignals` is set to `'on'`.<br>• Property name. S0, S1. . .<br>• Property value. Value of the signal.<br>• Block Name. Name of the Simulink block the signal is from. | |
| `S#` | Property name for a signal. | |
| `NumParameters` | The number of parameters from your Simulink model that you can tune or change. | |
| `ShowParameters` | Flag set to view or hide the list of parameters from your Simulink blocks. This list is shown when you display the properties for a target object. Values are `'on'` or `'off'`. | Yes |

| Property | Description | Write |
|---|---|---|
| `Parameters` | List of tunable parameters. This list is visible only when `ShowParameters` is set to `'on'`.<br>• Property name. P0, P1. . .<br>• Property value. Value of the parameter in a Simulink block.<br>• Type. Data type of the parameter. Always `double`.<br>• Size. Size of the parameter. For example, scalar, 1-by-2 vector, or 2-by-3 matrix.<br>• Parameter name. Name of a parameter in a Simulink block.<br>• Block name. Name of a Simulink block | Yes |
| `P#` | Property name for only tunable block parameter. One block can have one or more parameters. | |

## Target Object Methods

The target object methods allow you to control a target application on the target PC from the host PC. You enter target object methods in the MATLAB window on the host PC or use M-file scripts.

If you want to control the target application from the target PC, use target PC commands. See “Target PC Command-Line Interface” on page 1-31.

The target object methods are listed in the following table.

| Method | Description |
|---|---|
| `xpc` | Create a target object on the host PC (constructor). |
| `set` | Set writable target object properties to the specified value. |
| `get` | Return the value of readable properties from a target object. |
| `start` | Start the execution of a target application on the target PC. |
| `stop` | Stop the execution of a target application on the target PC. |
| `load` | Download a target application from the host PC to the target PC. |
| `unload` | Unload a target application from the target PC. If a target application is running, it is stopped and unloaded. |
| `addscope` | Create a new scope with type `'host'` or `'target'` on the target PC. |
| `getscope` | Return the properties of a previously created scope from the target PC. The scope properties can be assigned to a MATLAB variable to create a scope object. |

| Method | Description |
|---|---|
| `remscope` | Remove a scope from the target PC. This method does not remove the scope object, on the host PC, that represent the scope. |
| `getparamid` | Return the property name or index of a parameter from the target object. |
| `getsignalid` | Return the property name or index of a signal from the target object. |
| `getlog` | Upload and returns one of the data logs from the target PC to the host PC. TimeLog, StateLog, OutputLog, TETLog |
| `reboot` | Reboot the target PC. If a target application is running, the target application is stopped, and then the target PC is rebooted. |
| `close` | Close the serial connection to the target PC so that the host PC can use the COMM port for another device. |

# Using Target Objects

xPC Target uses a target object to represent the target kernel and your target application. This section shows some of the common tasks that you use with target objects. This section includes the following topics:

- Displaying Target Object Properties
- Setting Target Object Properties from the Host PC
- Setting Target Object Properties from the Target PC
- Getting the Value of a Target Object Property
- Using the Method Syntax with Target Objects

## Displaying Target Object Properties

You might want to list the target object properties to monitor a target application. The properties include the execution time, and average task execution time.

After you build a target application and target object from a Simulink model, you can list the target object properties. This procedure uses the default target object name `tg` as an example:

**1** In the MATLAB window, type

```
tg
```

The current target application properties are uploaded to the host PC, and MATLAB displays a list of the target object properties with the updated values.

**Note** the target object properties for TimeLog, StateLog, OutputLog, and TETLog are not updated at this time.

**2** Type

```
+tg
```

The Status property changes from `stopped` to `running`, and the log properties change to `Acquiring`.

For a list of target object properties with a description, see “Target Object Properties” on page 5-3.

## Setting Target Object Properties from the Host PC

You can change a target object property by using xPC Target methods on the host PC.

With xPC Target you can use either a function syntax or an object property syntax. The syntax `set(target_object, property_name, new_property_value)` can be replaced by

```
target_object.property_name = new_property_value
```

For example, to change the stop time mode for the target object `tg`:

1. In the MATLAB window, type

```
tg.stoptime = 1000
```

2. Alternatively, you could type

```
set(tg, 'stoptime', 1000)
```

Parameters are also target object properties. For example, to change the frequency of the signal generator in the model `xpcosc`:

1. In the MATLAB window, type

```
tg.p2 = 30
```

2. Alternatively, you could type

```
set(tg, 'p2', 30)
```

When you change a target object property, the new property value is downloaded to the target PC. The xPC Target kernel then receives the information and changes the behavior of the target application.

To get a list of the writable properties, type `set(target_object)`. The build process assigns the default name of the target object to `tg`.

---

**Note** Method names are case sensitive and need to be complete, but property names are not case sensitive and need not be complete as long as they are unique.

---

## Setting Target Object Properties from the Target PC

You can type commands directly from a keyboard on the target PC. These commands create a temporary difference between the behavior of the target application and the properties of the target object. The next time you access the target object, the properties are updated from the target PC:

1. On the target PC keyboard, press C, or point the target mouse in the command window.

   The target PC activates the command window.

2. Type a target command. For example, to change the frequency of the signal generator (parameter 2) in the model `xpcosc`, type

   ```
   setpar 2=30
   ```

3. Change the stop time. For example, to set the stop time to 1000, type

   ```
   stoptime = 1000
   ```

   The parameter changes are made to the target application but not to the target object. When you type any xPC Target command in the MATLAB command window, the target PC returns the current properties to the target object.

   **Note** The target PC command `setpar` does not work for vector parameters.

## Getting the Value of a Target Object Property

You can list a property value in the MATLAB window, or assign that value to a MATLAB variable. With xPC Target you can use either a function syntax or an object property syntax.

The syntax `get(`*`target_object`*`,`*`property_name`*`)` can be replaced by

```
target_object.property_name
```

For example, to access the start time:

1. In the MATLAB window, type

   ```
   endrun = tg.stoptime
   ```

2. Alternatively, you could type

   `endrun = get(tg,'stoptime')` or `tg.get('stoptime')`

Signals are also target object properties. For example, to get the value of the Integrator1 signal from the model xpcosc:

1. In the MATLAB window, type

   ```
   outputvalue= tg.S0
   ```

2. Alternatively, you could type

   `outputvalue = get(tg, 's2')` or `tg.get('s2')`

To get a list of readable properties, type *`target_object`*. Without assignment to a variable, the property values are listed in the MATLAB window.

---

**Note** Method names are case sensitive and need to complete, but property names are not case sensitive and need not be complete as long as they are unique.

---

## Using the Method Syntax with Target Objects

Use the method syntax to run a target object method. The syntax *method_name*(*target_object*, *argument_list*) can be replaced with

```
target_object.method_name(argument_list)
```

Unlike properties, for which partial but unambiguous names are permitted, method names must be entered in full, and in lowercase. For example, to add a scope of type `target` with a scope index of `1`:

1. In the MATLAB window, type

   ```
   tg.addscope('target',1)
   ```

2. Alternatively, you could type

   ```
   addscope(tg, 'target', 1)
   ```

## Purpose

Create one or more scopes on the target PC

## Syntax

**MATLAB command line**

Creating a scope and scope object without assigning to a MATLAB variable.

```
addscope(target_object, scope_type, scope_number)
target_object.addscope(scope_type, scope_number).
```

Creating a scope, scope object, and assign to a MATLAB variable.

```
scope_object = addscope(target_object, scope_type, scope_number)

scope_object = target_object.addscope(scope_type, scope_number)
```

**Target PC command line** — When using this command on the target PC, it is limited to adding a scope of type `target`.

```
addscope
addscope scope_number
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object. The default target name is `tg`. |
| *scope_type* | Values are `'host'` or `'target'`. This argument is optional with `host` as the default value. |
| *scope_number* | Vector of new scope indices. This argument is optional with the next available integer in the target object property `Scopes` as the default value.<br>If you enter a scope index for an existing scope object, the result is an error. |

## Description

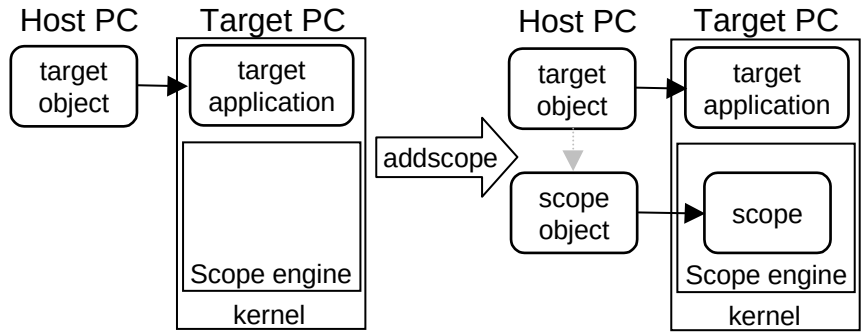
Method of a target object. Creates a scope on the target PC, a scope object on the host PC, and updates the target object property `Scopes`. This method returns a scope object vector. If the result is not assigned to a variable, the scope object properties are listed in the MATLAB window. If you try to add a scope with the same index as an existing scope, the result is an error.

A scope acquires data from the target application and displays that data on the target PC or uploads the data to the host PC.

All scopes of type `target` or `host` run on the target PC.

**Scope of type target** — Data collected is displayed on the target screen and acquisition of the next data package is initiated by the kernel.

**Scope of type host** — Collects data and waits for a command from the host PC for uploading the data. The data is then displayed using the host scope GUI (`xpcscope`) or other MATLAB functions.




## Examples

Create a scope and scope object `sc1` using the method `addscope`. A target scope is created on the target PC with an index of 1, a scope object is created on the host PC, and it is assigned to the variable `sc1`. The target object property `Scopes` is changed from `No scopes defined` to `1`.

```
sc1 = addscope(tg,'target',1)
```
or
```
sc1 = tg.addscope('target',1)
```

Create a scope with the method `addscope` and then to create a scope object, corresponding to this scope, using the method `getscope`. A target scope is created on the target PC with an index of 1, a scope object is created on the host PC, but it is not assigned to a variable. The target object property `Scopes` is changed from `No scopes defined` to `1`.

```
addscope(tg,'target',1)
```
or
```
tg.addscope('target',1)
sc1 = getscope(tg,1)
```
or
```
sc1 = tg.getscope(1)
```

Create two scopes using a vector of scope objects `scvector`. Two target scopes are created on the target PC with scope indices of 1 and 2, and two scope objects are created on the host PC that represent the scopes on the target PC. The target object property `Scopes` is changed from `No scopes defined` to `1,2`.

```
scvector = addscope(tg, 'target', [1, 2])
```

## See Also

The xPC Target target object methods `remscope`, `getscope`. The xPC Target GUI function `xpcscope`. The xPC target M-file demo scripts listed in “xPC Target Demos” on page 4-17.

# close

## Purpose

Close the serial port connecting the host PC with the target PC

## Syntax

**MATLAB command line**

```
close(target_object)
target_object.close
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object. |

## Description

Method of a target object. If you want to use the serial port for another function without quitting MATLAB, for example a modem, use this function to close the connection.

## Purpose

Return the property values for target and scope objects

## Syntax

**MATLAB command line**

```
get(target_object, 'target_object_property')
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object. |
| *target_object_property* | Name of a target object property. |

## Description

Method of target objects. Gets the value of readable target object properties from a target object.

## Examples

List the value for the target object property `StopTime`. Notice that the property name is a string, in quotation marks, and not case sensitive.

`get(tg,'stoptime')` or `tg.get('stoptime')`

```
ans = 0.2
```

## See Also

The xPC Target target object method `set`.The scope object methods `get` and `set`. The built in MATLAB functions `get` and `set`.

# getlog

## Purpose

Get all or part of the output logs from the target object

## Syntax

**MATLAB command line**

```
log = getlog(target_object, 'log_name', start_time,
number_points, decimation)
```

## Arguments

| | |
|---|---|
| *log* | User-defined MATLAB variable. |
| *log_name* | Values are `TimeLog`, `StateLog`, `OutputLog`, or `TETLog`. This argument is required. |
| *first_point* | First data point. The logs begin with 1. This argument is optional. Default is 1. |
| *number_points* | Number of points after the start time. This argument is optional. Default is all points in log. |
| *decimation* | 1 returns all sample points. n returns every nth sample point. This argument is optional. Default is 1. |

## Description

Method of a target object. Use this function instead of the function `get` when you want only part of the data.

## Examples

To get the first 1000 points in a log:

```
Out_log = getlog(tg, 'TETLog', 1, 1000)
```

To get every other point in the output log and plot values:

```
Output_log = getlog(tg, 'TETLog', 1, ,2)
Time_log = getlog(tg, 'TimeLog', 1, ,2)
plot(Time_log, Output_log)
```

## See Also

The xPC Target target object method `get`. The procedures “Entering the Real-Time Workshop Parameters” on page 3-26 and “Entering the Real-Time Workshop Parameters” on page 3-26.

## Purpose

Get a parameter index or property name from the parameter list

## Syntax

**MATLAB command line**

```
getparamid(target_object, 'block_name', 'parameter_name')
getparamid(target_object, 'block_name', 'parameter_name',
'flag')
```

## Arguments

| | |
|---|---|
| target_object | Name of a target object. The default name is `tg`. |
| *`block_name`* | Simulink block path and name. |
| *`parameter_name`* | Name of a parameter within a Simulink block. |
| *`flag`* | If `flag = property`, return the property name for the parameter. If `flag = numeric`, return a number index. This argument is optional. The default behavior is to return a property name. |

## Description

Method of a target object. Returns the index of a parameter in the parameter list based on the path to the parameter name. The names must be entered in full and are case sensitive.

## Examples

Get the parameter property for the parameter `Gain` in the Simulink block `Gain1`, incrementally increase gain, and pause to observe signal trace.

```
id = getparamid(tg, 'Subsystem/Gain1', 'Gain')
for i = 1 : 3
  set(tg, id, i*2000);
  pause(1);
end
```

Get the property name of a single block.

```
getparamid(tg, 'Gain1', 'Gain')
ans = P5
```

Get the parameter index of a single block.

```
getparamid(tg, 'Gain1', 'Gain', 'numeric')
ans = 5
```

P5 is a property of the target object. For example, you could assign a value to the gain with `tg.p5 = 1000`.

## See Also

The xPC Target scope object method `getsignalid`. The xPC target M-file demo scripts listed in “xPC Target Demos” on page 4-17.

## Purpose

Get a scope object pointing to a scope already defined in the kernel

## Syntax

**MATLAB command line**

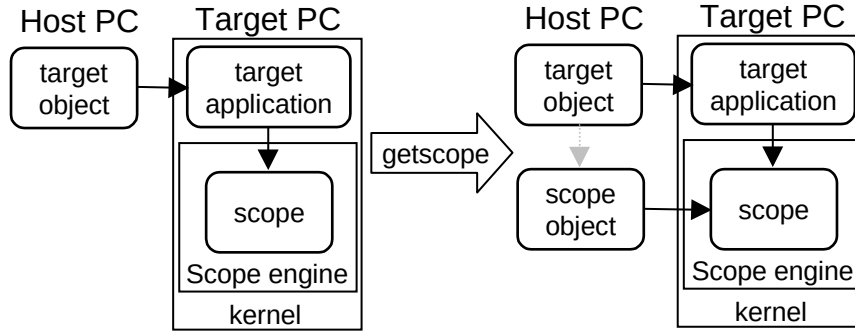
```
scope _object_vector = getscope(target_object, scope_number)
scope_object = target_object.getscope(scope_number)
```

## Arguments

| | |
|---|---|
| ***target_object*** | Name of a target object |
| *scope_number*_vector | Vector of existing scope indices listed in the target object property `Scopes`. The vector can have only one element. |
| *scope_object* | MATLAB variable for a new scope object vector. The vector can have only one scope object. |

## Description

Method of a target object. Returns a scope object vector. If you try to get a nonexistent scope, the result is an error. You can retrieve the list of existing scopes using the method `get(`*target_object*`, 'scopes')` or *target_object*`.scopes`.




## Examples

If your Simulink model has an xPC Target scope block, a scope of type `target` is created at the time the target application is downloaded to the target PC. To change the number of samples, you need to create a scope object and then change the scope object property `NumSamples`.

```
sc1 = getscope(tg,1)  or sc1 = tg.getscope(1)
sc1.NumSample = 500
```

To get the properties of all scopes on the target PC and create a vector of scope objects on the host PC. If the target object has more than one scope, creates a vector of scope objects.

```
scvector = getscope(tg)
```

## See Also

The xPC Target target object methods `addscope` and `remscope`. The xPC target M-file demo scripts listed in “xPC Target Demos” on page 4-17.

## Purpose

Get the signal index or signal property from the signal list

## Syntax

**MATLAB command line**

```
getsignalid(target_object, 'signal_name')
getsignalid(target_object, 'signal_name', 'flag')
tg.getsignalid('signal_name')
```

## Arguments

| | |
|---|---|
| *target_object* | Name of an existing target object. |
| *signal_name* | Enter the name of a signal from your Simulink model. For blocks with a single signal, the *signal_name* is equal to the *block_name*. For blocks with multiple signals, xPC Target appends S1, S2 ... to the *block_name*. |
| *flag* | If `flag = property`, return the property name for the signal. If `flag = numeric`, return a number index. This argument is optional. The default behavior is to return a number. |

## Description

Method of a target object. Returns the index or name of a signal from the signal list, based on the path to the signal name. The block names must be entered in full and are case sensitive.

## Examples

Get the signal property for the single signal from the Simulink block `Gain1`.

`getsignalid(tg, 'Gain1')` or `tg.getsignal('Gain1')`

```
ans = S6
```

Get the signal index for the single signal from the Simulink block `Gain1`.

```
getsignalid(tg, 'Gain1', 'numeric')
ans = 6
```

The signal index S6 is also a property of the target object. For example, you could get the value of a signal with `signal_6 = tg.s6`.

If a Simulink block has two or more signal outputs, you need to the extended block name. For example, if a block is labled `Gain` has two signals, the signal names would be `Gain/S1` and `Gain/S2`. Get the signal index for a multiple signal from the Simulink block `Gain`.

```
getsignalid(tg, 'Gain/S1', 'numeric'
ans = 3
```

**See Also**

The target object method `getparamid`. The xPC Target M-file demo scripts listed in “xPC Target Demos” on page 4-17.

## Purpose

Download a target application to the target PC

## Syntax

**MATLAB command line**

```
load(target_object,'target_application')
target_object.load('target_application')
```

## Arguments

| | |
|---|---|
| *target_object* | Name of an existing target object. |
| *target_application* | Simulink model and target application name. |

## Description

Method of a target object. Before using this function, the target PC must be booted with the xPC Target kernel, and the target application must be built in the current working directory on the host PC.

If an application was previously loaded, the old target application is first unloaded before downloading the new target application. The method `load` is called automatically after the Real-Time Workshop build process.

## Examples

Load the target application `xpcosc` represented by the target object `tg`.

`load(tg,'xpcosc')` or `tg.load('xpcosc')`
`+tg` or `tg.start` or `start(tg)`

## See Also

The xPC Target function `unload`. The xPC target M-file demo scripts listed in "xPC Target Demos" on page 4-17.

# reboot

## Purpose

Reboot the target PC

## Syntax

**MATLAB command line**

```
reboot(target_object)
```

**Target PC command line**

```
reboot
```

## Arguments

| | |
|---|---|
| *target_object* | Name of an existing target object. |

## Description

Method of a target object. Reboots the target PC, and if a target boot disk is still present, the xPC target kernel is reloaded.

You can also use this method to reboot the target PC back to Windows after removing the target boot disk.

**Note** This method might not work on some target hardware.

## See Also

The xPC Target target object methods `load` and `unload`.

## Purpose

Remove a scope from the target PC

## Syntax

**MATLAB command line**

```
remscope(target_object, scope_number_vector)
target_object.remscope(scope_number_vector)

remscope(target_object)
target_object.remscope
```

**Target PC command line**

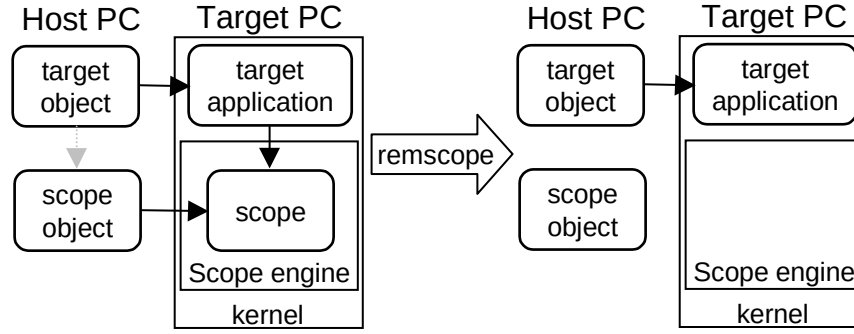
```
remscope scope_number
remscope 'all'
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object. The default name is `tg`. |
| *scope_number*_vector | Vector of existing scope indices listed in the target object property `Scopes`. |
| *scope_number* | Single scope index. |

## Description

Method of a target object. If a scope index is not given, the method remscope deletes all scopes on the target PC. The method `remscope` has no return value. The scope object representing the scope on the host PC is not deleted.



## Examples

Remove a single scope.

```
remscope(tg,1) or tg.remscope(1)
```

Remove two scopes.

```
remscope(tg,[1 2]) or tg.remscope([1,2])
```

Remove all scopes.

```
remscope(tg) or tg.remscope
```

## See Also

The xPC Target target object methods `addscope` and `getscope`. The xPC target M-file demo scripts listed in “xPC Target Demos” on page 4-17.

## Purpose

Change property values for target objects

## Syntax

**MATLAB command line**

```
set(target_object)

set(target_object, property_name1, property_value1,
property_name2, property_value2, . . .)

target_object.set('property_name1', property_value1)

set(target_object, property_name_vector, property_value_vector)

target_object.property_name = property_value
```

**Target PC command line** - Commands are limited to the target object properties: `stoptime`, `sampletime`, and parameters.

```
parameter_name = parameter_value
stoptime = floating_point_number
sampletime = floating_point_number
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object. |
| '*property_name*' | Name of a scope object property. Always use quotation marks. |
| *property_value* | Value for a scope object property. Always use quotation marks for character strings; quotation marks are optional for numbers. |
| *parameter_name* | The letter p followed by the parameter index. For example, `p0`, `p1`, `p2`. |

## Description

Method of a target object. Sets the properties of the target object. Not all properties are user-writable.

Properties must be entered in pairs, or using the alternate syntax, as one-dimensional cell arrays of the same size. This means they have to both be row vectors or both column vectors, and the corresponding values for properties in *property_name_vector* are stored in *property_value_vector*.

The function set typically does not return a value. However, if called with an explicit return argument, for example, `a = set(target_object, property_name, property_value)`, it returns the value of the properties after the indicated settings have been made.

## Examples

Get a list of writable properties for a scope object.

```
sc1 = getscope(tg,1)
set(sc1)

xPC Target Object:
    Writable Properties

    StopTime
    SampleTime
    ViewMode
    LogMode             : [0 | 1]
    ShowParameters      : [On | {Off}]
    ShowSignals         : [On | {Off}]
```

Change the property showsignals to on.

```
tg.set('showsignals', 'on') or set(tg, 'showsignals', 'on')
```

As an alternative to the method set, use the target object property showsignals. In the MATLAB window, type

```
tg.showsignals ='on'
```

## See Also

The xPC Target target object method get. The scope object methods get and set. The built in MATLAB functions get and set. The xPC target M-file demo scripts listed in “xPC Target Demos” on page 4-17.

## Purpose

Start execution of a target application on a target PC

## Syntax

**MATLAB command line**

```
start(target_object)
target_object.start
+target_object
```

**Target PC command line**

```
start
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object. The default name is `tg`. |

## Description

Method of both target and scope objects. Starts execution of the target application represented by the target object. Before using this method, the target application must be created and loaded on the target PC. If a target application is running, this command has no effect.

## Examples

Start the target application represented by the target object `tg`.

```
+tg
tg.start
start(tg)
```

## See Also

The xPC Target target object methods `stop` on page 5-36, `load` on page 5-29, and `unload` on page 5-37. The scope object method `stop` on page 6-24.

# stop

## Purpose

Stop execution of a target application on a target PC

## Syntax

**MATLAB command line**

```
stop(target_object)
target_object.stop
-target_object
```

**Target PC command line**

```
stop
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object. |

## Description

Stops execution of the target application represented by the target object. If the target application is stopped, this command has no effect.

## Examples

Stop the target application represented by the target object `tg`.

```
stop(tg) or tg.stop or -tg
```

## See Also

The xPC Target target object method `start` on page 5-35. The scope object methods `stop` on page 6-24 and `start` on page 6-22.

## Purpose

Remove the current target application from the target PC

## Syntax

**MATLAB command line**

```
unload(target_object)
target_object.unload
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object that represents a target application. |

## Description

Method of a target object. The kernel goes into loader mode and is ready to download new target application from the host PC.

## Examples

Unload the target application represented by the target object `tg`.

`unload(tg)` or `tg.unload`

## See Also

The xPC Target methods `load` and `reboot`.

## Purpose

Create a target object representing the target application

## Syntax

**MATLAB command line**

```
target_object = xpc
```

## Arguments

| | |
|---|---|
| *target_object* | Variable name to reference the target object. |

## Description

Constructor of a target object. The target object represents the target application and target PC. Changes are made to the target application by making changes to the target object, using methods and properties.

## Examples

Before you build a target application, you can check the connection between your host and target computers by creating a target object.

```
tg = xpc
```

```
xPC Object

   Connected            = Yes
   Application          = loader
```

## See Also

The xPC Target methods `get` on page 5-21 and `set` on page 5-33.

6

# Scope Objects

xPC Target uses scope objects to represent the data acquisition part of the xPC Target kernel. Use scope objects to view and collect signal data. This chapter includes the following sections:

# Scope Object Reference

xPC Target uses scopes and scope objects as an alternative to using Simulink scopes and external mode. Understanding the structure of scope objects will help you to use the MATLAB command-line interface to view and collect signal data. The topics in this section are

- **What Is a Scope Object?** — Definition, and ways to create scope objects
- **Scope Object Properties** — List of properties with definitions
- **Scope Object Methods** — List of methods with definitions

## What Is a Scope Object?

A scope object on the host PC represents a scope on the target PC. You use scope objects to observe the signals from your target application during a real-time run or analyze the data after the run is finished.

To create a scope object:

- Add an xPC Target scope block to your Simulink model, build the model to create a scope, and then use the target object method `getscope` to create a scope object.
- Use the target object method `addscope` to create a scope, create a scope object and assign the scope properties to the scope object.

A scope object has associated properties and methods specific to that object.

## Scope Object Properties

Scope object properties let you select signals to acquire, set triggering modes, and access signal information from the target application. You can view and change these properties using scope object methods

The properties for a scope object are listed in the following table. This table includes a description of the properties and which properties you can change directly by assigning a value.

| Property | Description | Write |
|---|---|---|
| `Application` | Name of the Simulink model associated to this scope object. | |
| `ScopeId` | A numeric index unique for each scope. | |
| `Status` | Indicate whether data is being acquired, the scope is waiting for a trigger, the scope has been stopped (interrupted), or acquisition is finished. Values are `'Acquiring'`, `'Ready for being Triggered'`, `'Interrupted'`, and `'Finished'`. | |
| `Type` | Determine whether the scope is displayed on the host computer or on the target computer. Values are `'host'` and `'target'`. | |
| `NumSamples` | Number of contiguous samples captured during the acquisition of a data package. | Yes |
| `NumPrePostSamples` | Number of samples collected before or after a trigger event. The default value is 0. Entering a negative value collects samples before the trigger event. Entering a positive value collects samples after the trigger event. If you set `TriggerMode` to `FreeRun`, this property has no effect on data acquisition. | |

| Property | Description | Write |
|---|---|---|
| `Decimation` | A number `n`, where every nth sample is acquired in a scope window.<br><br>**Note** This value is the same as **Interleave** in a scope window. | Yes |
| `TriggerMode` | Trigger mode for a scope. Valid values are `'FreeRun'` (default), `'Software'`, `'Signal'`, and `'Scope'`. | Yes |
| `TriggerSignal` | If `TriggerMode='Signal'`, identifies which block output signal to use for triggering the scope. You identify the signal with a signal index from the target object property `Signal`. | Yes |
| `TriggerSample` | If `TriggerMode='Scope'`, then `TriggerSample` specifies which sample of the triggering scope the current scope should trigger on. For example, if `TriggerSample = 0` (default), the current scope will trigger on sample `0` (first sample acquired) of the triggering scope. This means that the two scopes will be perfectly synchronized. If `TriggerSample = 1`, the first sample (sample `0`) of the current scope will be the at the same instant as sample number `1` (second sample in the acquisition cycle) of the triggering scope.<br><br>As a special case, setting TriggerSample to `-1` means that the current scope will be triggered at the end of the acquisition cycle of the triggering scope. Thus, the first sample acquired of the triggering scope will be the one sample after the last sample of the triggering scope. | |

| Property | Description | Write |
|---|---|---|
| `TriggerLevel` | If `TriggerMode='Signal'`, indicates the value the signal has to cross to trigger the scope and start acquiring data. The trigger level can be crossed with either a rising or falling signal. | Yes |
| `TriggerSlope` | If `TriggerMode='Signal'`, indicates whether the trigger is on a rising or falling signal. Values are `'Either'` (default), `'Rising'`, or `'Falling'`. | Yes |
| `TriggerScope` | If `TriggerMode='Scope'`, identifies which scope to use for a trigger. A scope can be set to trigger when another scope is triggered. This is done by setting the slave scope property `TriggerScope` to the scope index of the master scope. | Yes |
| `Mode` | Indicate how a scope displays the signals. Values are `'Numerical'`, `'Redraw'` (default), `'Sliding'`, or `'Rolling'`. | Yes |
| `YLimit` | Minimum and maximum *y*-axis values. This property can be set to `'auto'`. | Yes |
| `Grid` | Values are `'on'` or `'off'`. | Yes |
| `StartTime` | Time within the total execution time, when a scope begins acquiring a data package. | |
| `Time` | Contains the time data for a single data package from a scope. | |
| `Data` | Contains the output data for a single data package from a scope. | |
| `Signals` | List of signal indices from the target object to display on the scope. | |

## Scope Object Methods

The scope object methods allow you to control scopes on your target PC.

If you want to control the target application from the target PC, use target PC commands. See “Target PC Command-Line Interface” on page 1-31.

The scope object methods are listed in the following table.

| Scope Method | Description |
|---|---|
| `set` | Set writable scope object properties to the specified value. For a list of writable values, use `set(scope_object)` |
| `get` | Return the value of readable properties from a scope object. |
| `addsignal` | Add a signal to a scope and a scope object. The signal is specified using the signal indices from the target object. |
| `remsignal` | Remove a signal from a scope and a scope object. The signal is specified using signal indices from the scope object. |
| `start` | Start a scope, but does not necessarily start the acquisition of data. The acquisition of data is dependent on the trigger mode. |
| `stop` | Stop a scope and the acquisition of data. |
| `trigger` | Starts the acquisition of data from the target application using a scope.<br>• If `TriggerMode ='Software'`, then the scope can only be triggered by software using the method `sc.trigger`<br>• If TriggerMode = `'FreeRun'`, `'Signal'`, or `'Scope'`, you can trigger the scope by software before the scope is triggers by one of the other modes. |

# Using Scope Objects

xPC Target uses scope objects to represent scopes on the target PC. This section shows some of the common tasks that you use with scope objects. The topics in this section are

- Displaying Scope Object Properties for a Single Scope
- Displaying Scope Object Properties for All Scopes
- Setting the Value of a Scope Property
- Getting the Value of a Scope Property
- Using the Method Syntax with Scope Objects
- Using the Property TriggerSample to Capture Data

## Displaying Scope Object Properties for a Single Scope

To list the properties of a single scope object `sc1`:

**1** In the MATLAB window, type

```
sc1  = getscope(tg,1) or sc1 = tg.getscopes(1)
```

MATLAB creates the scope object `sc1` from a previously created scope.

**2** Type

```
sc1
```

The current scope properties are uploaded to the host PC, and then MATLAB displays a list of the scope object properties with the updated values. Because `sc1` is a vector with a single element, you could also type `sc1(1)` or `sc1([1])`.

---

**Note** Only scopes with type `host` store data in the properties `scope_object.Time and scope_object.Data`.

---

For a list of target object properties with a description, see “Target Object Properties” on page 5-3.

## Displaying Scope Object Properties for All Scopes

To list the properties of all scope objects associated with the target object `tg`:

1. In the MATLAB window, type

   ```
   getscope(tg) or tg.getscope
   ```

   MATLAB displays a list of all scope objects associated with the target object.

2. Alternatively, type

   ```
   allscopes = getscope(tg)
   ```

   or type

   ```
   allscopes = tg.getscope
   ```

   The current scope properties are uploaded to the host PC, and then MATLAB displays a list of all the scope object properties with the updated values. To list some of the scopes, use the vector index. For example, to list the fist and third scopes, type `allscopes([1,3])`.

For a list of target object properties with a description, see "Target Object Properties" on page 5-3.

## Setting the Value of a Scope Property

With xPC Target you can use either a function syntax or an object property syntax. The syntax `set(scope_object, property_name, new_property_value)` can be replaced by

- `scope_object_vector.property_name = new_property_value.`
- `scope_object(index_vector).property_name = new_property_value.`

For example, to change the trigger mode for the scope object `sc1`:

1. In the MATLAB window, type

   ```
   sc1.triggermode = 'signal'
   ```

2. Alternatively, you could type

   ```
   set(sc1,'triggermode', 'signal')
   ```

   or type

   ```
   sc1.set('triggermode', 'signal')
   ```

Assignment can also be done for a vector of scope objects, for example `allscopes([1, 2]).numsamples = 500`. Notice, the indices are MATLAB vector indices and not xPC Target scope indices.

To get a list of the writable properties, type `set(scope_object)`.

---

**Note** Method names are case sensitive, but property names are not.

---

## Getting the Value of a Scope Property

You can list a property value in the MATLAB window, or assign that value to a MATLAB variable. With xPC Target you can use either a function syntax or an object property syntax.

The syntax `get(scope_object_vector, property_name)` can be replaced by

- `scope_object_vector.property_name`
- `scope_object_vector(index_vector).property_name`

For example, to assign the start time from the scope object `sc1`:

1. In the MATLAB window, type

   ```
   beginrun = sc1.starttime
   ```

2. Alternatively, you could type

   ```
   beginrun = get(sc1,'starttime')
   ```

   or type

   ```
   sc1.get('starttime')
   ```

Assignment can also be done using a vector of scope objects, for example

```
scopetypes = allscopes([1, 2]).type
```

Notice that the indices are MATLAB vector indices and not xPC Target scope indices.

To get a list of readable properties, type `scope_object`. The property values are listed in the MATLAB window.

**Note** Method names are case sensitive, but property names are not.

## Using the Method Syntax with Scope Objects

Use the method syntax to run a scope object method. The syntax `method_name(scope_object_vector, argument_list)` can be replaced with

- `scope_object.method_name(argument_list)`
- `scope_object_vector(index_vector).method_name(list of arguments)`

Unlike properties, for which partial but unambiguous names are permitted, method names must be entered in full, and in lowercase. For example, to add signals to the first scope in a vector of all scopes:

**1** In the MATLAB window, type

```
allscopes(1).addsignal([0,1])
```

**2** Alternatively, you could type

```
addsignal(allscopes(1), [0,1])
```

## Using the Property TriggerSample to Capture Data

xPC Target scopes include the ability to have one scope trigger another, and to delay retrieving data from the second after a trigger event on the first. This feature is most useful when data acquisition for the second scope is triggered after data acquisition for the first scope is complete. In the following explanation, Scope 2 is triggered by Scope 1:

- The two scopes objects are created as a vector with the command

```
sc = tg.addscope('host', [1 2]);
```

- For Scope 1, sc(1).ScopeId = 1, sc(1).NumSamples = N, sc1.NumPrePost Samples = *P*
- For Scope 2, `sc(2).ScopeId = 2`, `sc(2).TriggerMode = 'Scope'`, `sc(2).TriggerScope =1`, `sc(2).TriggerSample =` range `0` to (*N* `+` *P* `- 1`).

In the figures below, TP is the trigger point or sample where a trigger event occurs.

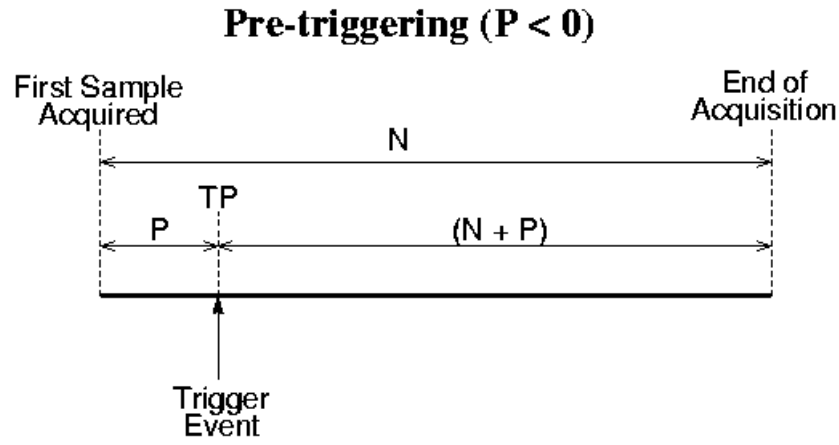



**Figure 6-1: Pre-triggering (P<0)**

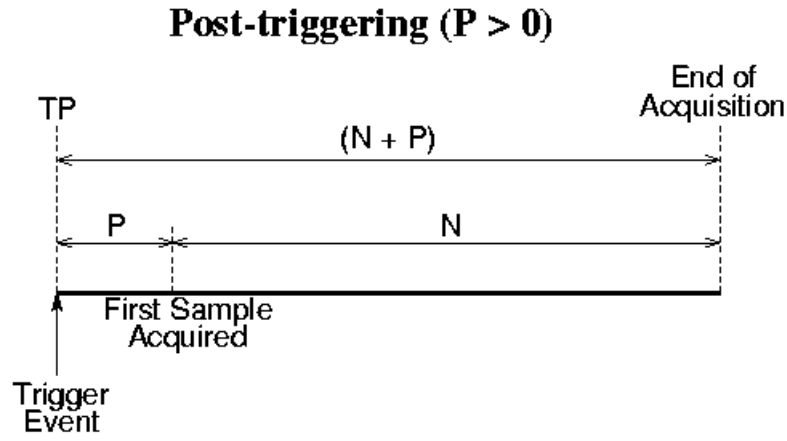



**Figure 6-2: Post-triggering (P>0)**

In the simplest case, where `P = 0`, the first sample will be acquired. If `P < 0` (pre-triggering), acquisition will start `|P|` samples before `TP`, and if `P > 0` (post-triggering), acquisition will start `P` samples after `TP`. It is not possible for Scope 1 to trigger Scope 2 before the trigger event occurs:

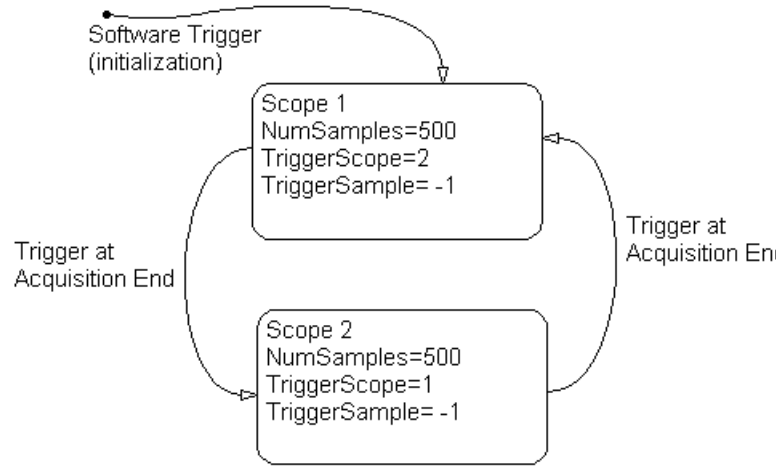
- Setting `sc(2).TriggerSample = 0` means that Scope 2 will be triggered when Scope 1 is triggered. `TP` for both scopes will be at the same sample.
- Setting `sc(2).TriggerSample = n > 0` means that `TP` for Scope 2 will be `n` samples after `TP` for Scope 1. Note that setting `sc(2).TriggerSample` to a value larger than `(N + P - 1)` does not cause an error; it however implies that Scope 2 will never trigger, since Scope 1 will never acquire more than `(N + P - 1)` samples after `TP`.
- Setting the `sc(2).TriggerSample = 0 < n < (N + P)` enables you to obtain some of the functionality that is available with pre- or post-triggering. For example, if Scope 1 has `sc(1).NumPrePostSamples = 0` (no pre- or post-triggering), and Scope 2 has `sc(2).TriggerSample = 10` and `sc(2).NumPrePostSample = 0`, the behavior displayed by Scope 2 is equivalent to having `sc(2).TriggerSample = 0` and `sc(2).NumPrePostSamples = 10`.

**Note** The difference between setting `TriggerSample = (N + P - 1)`, where `N` and `P` are the parameters of the triggering scope (Scope 1) and `TriggerSample = -1` is that in the former case, the first sample of Scope 2 will be at the same time as the last sample of Scope 1, whereas in the latter, the first sample of Scope 2 will be one sample after the last sample of Scope 1. This means that in the former case both scopes will be acquiring simultaneously for one sample, and in the latter they will never be simultaneously acquiring.

### Acquiring Gap-Free Data

When `sc(1).TriggerSample = -1` and `sc(2).TriggerSample = -1` you get new functionality that cannot be achieved with pre- and post-triggering. This new functionality is gap-free acquisition of data.

To do this, you would need to set up two scopes, each triggered by the other, with both scopes having `TriggerSample = -1`. This is represented by the following figure.




Both the scopes receive exactly the same signals, i.e. the signals you wish to retrieve.

One of the scopes needs to be software triggered in order for acquisition to start. Otherwise, each scope would be waiting for the other to finish acquiring data, and would never start. In the example, Scope 1 is software triggered to start the acquisition.

The following script is a typical example of how you can use this feature to retrieve data.

```
% assumes that model is built and loaded on target.
tg = xpc;
sc = tg.addscope('host', [1 2]);
sc.addsignal([0 1]);  % [0 1] are the signals of interest; add
to both
% default value for TriggerSample is 0, need to change it.
set(sc, 'NumSamples', 500, 'TriggerSample', -1)
sc(1).TriggerScope = 2;
sc(2).TriggerScope = 1;
start(sc);
start(tg);
sc(1).trigger;       % start things off by triggering scope 1
data = zeros(0, 2);
t    = [];
scNum = 1;           % we will look at scope 1 first

% Use some appropriate condition instead of an infinite loop
while(1)
  % loop until the scope has finished
  while ~strcmp(sc(scNum).Status, 'Finished'), end
  data(end + 1 : end + 500, :) = sc(scNum).Data;
  t(   end + 1 : end + 500)    = sc(scNum).Time;
  start(sc(scNum));         % restart the scope
  scNum = 2 - scNum;        % switch to the next scope
end
```

This example assumes that the communication speed and number of samples is fast enough to acquire the full data set before the next acquisition cycle is due to start. You can also use more than two scopes to implement a triple- or quadruple-buffering scheme instead of the double-buffering one shown here.

# addsignal

## Purpose

Add signals to a scope represented by a scope object

## Syntax

**MATLAB command line**

```
addsignal(scope_object_vector, signal_index_vector)
scope_object_vector.addsignal(signal_index_vector)
```

**Target command line**

```
addsignal scope_index = signal_index, signal_index, . . .
```

## Arguments

| | |
|---|---|
| *scope_object_vector* | Name of a single scope object, or the name of a vector of scope objects. |
| *signal_index_vector* | For one signal, use a single number. For two or more signals, enclose numbers in brackets and separate with commas. |
| *scope_index* | Single scope index. |

## Description

Method of a scope object. The signals must be specified by their indices, which can be retrieved using the target object method `getsignalid`. If the *scope_object_vector* has two or more scope objects, the same signals are assigned to each scope.

## Examples

Add signals `0` and `1` from the target object `tg` to the scope object `sc1`. The signals are added to the scope, and the scope object property `Signals` is updated to include the added signals.

```
sc1 = getscope(tg,1)
addsignal(sc1,[0,1]) or sc1.addsignal([0,1])
```

Display a list of properties and values for the scope object `sc1` with the property `Signals`, as shown below.

```
sc1.Signals

Signals              = 1  : Signal Generator
                       0  : Integrator1
```

Another way to add signals without using the method `addsignal` is to use the scope object method `set`.

`set(sc1,'Signals', [0,1])` or `sc1.set('signals',[0,1]`

Or to directly assign signal values to the scope object property `Signals`.

```
sc1.signals = [0,1]
```

## See Also

The xPC Target scope object methods `remsignal` and `set`. The target object method `addscope` and `getsignalid`.

## Purpose

Return the property values for scope objects

## Syntax

**MATLAB command line**

```
get(scope_object_vector)
get(scope_object_vector, 'scope_object_property')
get(scope_object_vector, scope_object_property_vector)
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object. |
| *scope_object_vector* | Name of a single scope, or name of a vector of scope objects. |
| *scope_object_property* | Name of a scope object property. |

## Description

Method of a scope object. Gets the value of readable scope object properties from a scope object or the same property from each scope object in a vector of scope objects.

## Examples

List all of the readable properties, along with their current values. This is given in the form of a structure, whose field names are the property names and field values are property values.

```
get(sc)
```

List the value for the scope object property `Type`. Notice that the property name is a string, in quotes, and is not case sensitive.

```
get(sc,'type')
ans = Target
```

## See Also

The xPC Target scope object method `set`. The target object methods `get` and `set`. The built in MATLAB functions `get` and `set`.

## Purpose

Remove signals from a scope represented by a scope object

## Syntax

**MATLAB command line**

```
remsignal(scope_object)
remsignal(scope_object, signal_index_vector)
scope_object.remsignal(signal_index_vector)
```

**Target command line**

```
remsignal scope_index = signal_index, signal_index, . . .
```

## Arguments

| | |
|---|---|
| *scope_object* | MATLAB object created with the target object method `addscope` or `getscope`. |
| *signal_index_vector* | Index numbers from the scope object property `Signals`. This argument is optional, and if it is left out all signals are removed. |
| *signal_index* | Single signal index. |

## Description

Method for a scope object. The signals must be specified by their indices, which can be retrieved using the target object method `getsignalid`. If the *scope_object_vector* has two or more scope objects, the same signals are removed from each scope. The argument *signal_index* is optional; if it is left out, all signals are removed.

## Examples

Remove signals 0 and 1 from the scope represented by the scope object sc1.

```
sc1.get('signals')

ans= 0 1
```

Remove signals from the scope on the target PC with the scope object property `Signals` updated.

`remsignal(sc1,[0,1])` or `sc1.remsignal([0,1])`

## See Also

The xPC Target scope object method `remsignal`, and the target object method `getsignalid`.

## Purpose

Change property values for scope objects

## Syntax

**MATLAB command line**

```
set(scope_object_vector)

set(scope_object_vector, property_name1, property_value1,
property_name2, property_value2, . . .)

scope_object_vector.set('property_name1', property_value1, ..)

set(scope_object, 'property_name', property_value, . . .)
```

## Arguments

| | |
|---|---|
| *scope_object* | Name of a scope object, or a vector of scope objects. |
| *'property_name'* | Name of a scope object property. Always use quotation marks. |
| *property_value* | Value for a scope object property. Always use quotation marks for character strings; quotes are optional for numbers. |

## Description

Method for scope objects. Sets the properties of the scope object. Not all properties are user-writable

Properties must be entered in pairs, or using the alternate syntax, as one-dimensional cell arrays of the same size. This means they have to both be row vectors or both column vectors, and the corresponding values for properties in `property_name_vector` are stored in `property_value_vector`.

The function set typically does not return a value. However, if called with an explicit return argument, for example, `a = set(target_object, property_name, property_value)`, it returns the value of the properties after the indicated settings have been made.

## Examples

Get a list of writable properties for a scope object.

```
sc1 = getscope(tg,1)
set(sc1)

xPC Scope Object:
    Writable Properties
```

```
NumSamples
Decimation
TriggerMode   : [{FreeRun} | Software | Signal | Scope]
TriggerSignal
TriggerLevel
TriggerSlope  : [{Either} | Rising | Falling]
TriggerScope
Signals
Mode          : [Numerical | {Redraw} | Sliding | Rolling]
YLimit
Grid
```

The property value for the scope object sc1 is changed to on:

```
sc1.set('grid', 'on') or set(sc1, 'grid', 'on')
```

## See Also

The xPC Target scope object method `get`. The target object methods `set` and `get`. The built-in MATLAB functions `get` and `set`.

## Purpose

Start execution of a scope on a target PC

## Syntax

**MATLAB command line**

```
start(scope_object_vector)
scope_object_vector.start
+scope_object_vector

start(getscope((target_object, signal_index_vector))
```

**Target PC command line**

```
startscope scope_index
startscope 'all'
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object. |
| *scope_object_vector* | Name of a single scope object, name of vector of scope objects, list of scope object names in a vector form `[scope_object1, scope_object2]`, or the target object method `getscope`, which returns a `scope_object` vector. |
| *signal_index_vector* | Index for a single scope, or list of scope indices in vector form. |
| *scope_index* | Single scope index. |

## Description

Method for a scope object. Starts a scope on the target PC represented by a scope object on the host PC. This method does not necessarily start data acquisition, which depends on the trigger settings. Before using this method, a scope must be created. To create a scope, use the target object method `addscope` or add xPC Target scope blocks to your Simulink model.

## Examples

Start one scope with the scope object sc1.

```
sc1 = getscope(tg,1) or sc1 = tg.getscope(1)
start(sc1) or sc1.start or +sc1
```

or type

```
start(getscope(tg,1))
```

Start two scopes.

```
somescopes = getscope(tg,[1,2]) or somescopes=
tg.getscope([1,2])
start(somescopes) or somescopes.start
```

or type

```
sc1 = getscope(tg,1) or sc1 =tg.getscope(1)
sc2 = getscope(tg,2) or sc2 = tg.getscope(2)
start([sc1,sc2])
```

or type

```
start(getscope(tg,[1,2])
```

Start all scopes:

```
allscopes = getscope(tg) or allscopes = tg.getscope
start(allscopes) or allscopes.start or +allscopes
```

or type

```
start(getscope(tg)) or start(tg.getscope)
```

## See Also

The xPC Target target object methods `getscope` and `stop`. The scope object method `stop`.

# stop

## Purpose

Stop execution of a scope on the target PC

## Syntax

**MATLAB command line**

```
stop(scope_object_vector)
scope_object.stop
-scope_object
```

```
stop(getscope(target_object, signal_index_vector))
```

**Target PC command line**

```
stopscope scope_index
stopscope 'all'
```

## Arguments

| | |
|---|---|
| *target_object* | Name of a target object. |
| *scope_object_vector* | Name of a single scope object, name of vector of scope objects, list of scope object names in a vector form `[scope_object1, scope_object2]`, or the target object method `getscope`, which returns a `scope_object` vector. |
| *signal_index_vector* | Index for a single scope, or list of scope indices in vector form. |
| *scope_index* | Single scope index. |

## Description

Method for scope objects. Stops the scopes represented by the scope objects.

## Examples

Stop one scope represented by the scope object `sc1`.

`stop(sc1)` or `sc1.stop` or `-sc1`

Stop all scopes with a scope object vector `allscopes` created with the command `allscopes = getscope(tg)` or `allscopes = tg.getscope`.

`stop(allscopes)` or `allscopes.stop` or `-allscopes`

or type

`stop(getscope(tg))` or `stop(tg.getscope)`

## See Also

The xPC Target target object methods `getscope`, `stop`, and `start`. The scope object method `start`.

# trigger

## Purpose

Software-trigger the start of data acquisition for one or more scopes

## Syntax

**MATLAB command line**

`trigger(`*`scope_object_vector`*`)` or *`scope_object_vector`*`.trigger`

## Arguments

| | |
|---|---|
| *`scope_object_vector`* | Name of a single scope object, name of a vector of scope objects, list of scope object names in a vector form `[scope_object1, scope_object2]`, or the target object method `getscope`, which returns a `scope_object` vector. |

## Description

Method for a scope object. If the scope object property `TriggerMode` has a value of `'software'`, this function triggers the scope represented by the scope object to acquire the number of data points in the scope object property `NumSamples`.

**Note** Only scopes with type `host` store data in the properties `scope_object.Time` and `scope_object.Data`.

## Examples

Set a single scope to software trigger, trigger the acquisition of one set of samples, and plot data.

```
sc1 = tg.addscope('host',1) or sc1=addscope(tg,'host',1)
sc1.triggermode = 'software'

tg.start, or start(tg), or +tg
sc1.start  or start(sc1) or +sc1
sc1.trigger or trigger(sc1)

plot(sc1.time, sc1.data)

sc1.stop or stop(sc1) or -sc1
tg.stop or stop(tg) or -tg1
```

Set all scopes to software trigger and trigger to start.

```
allscopes = tg.getscopes
allscopes.triggermode = 'software'
allscopes.start or start(allscopes) or +allscopes
allscopes.trigger or trigger(allscopes)
```

# Index

## P



## S



## T



## U



## X