xPC Target 3

API Guide

MATLAB® & SIMULINK®

The MathWorks

Accelerating the pace of engineering and science

How to Contact The MathWorks

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

xPC Target API Guide

Revision History

July 2002	Online only	New for Version 2 (Release 13)
October 2002	Online only	Updated for Version 2 (Release 13)
September 2003	Online only	Revised for Version 2.0.1 (Release 13SP1)
June 2004	Online only	Revised for Version 2.5 (Release 14)
August 2004	Online only	Revised for Version 2.6 (Release 14+)
October 2004	Online only	Revised for Version 2.6.1 (Release 14SP1)
November 2004	Online only	Revised for Version 2.7 (Release 14SP1+)
March 2005	Online only	Revised for Version 2.7.2 (Release 14SP2)
September 2005	Online only	Revised for Version 2.8 (Release 14SP3)
March 2006	Online only	Revised for Version 2.9 (Release 2006a)
May 2006	Online only	Revised for Version 3.0 (Release 2006a+)
September 2006	Online only	Revised for Version 3.1 (Release 2006b)
March 2007	Online only	Revised for Version 3.2 (Release 2007a)

Contents

1 Introduction

2 xPC Target API

3 xPC Target COM API

4 xPC Target COM API Demos and Scripts

5 API Functions and Methods — By Category

6 API Functions and Methods — Alphabetical List

A xPC Target C API Error Messages

Index

Introduction

Using either the xPC Target API dynamic link library (DLL) or the xPC Target component object model (COM) API library, you can create custom applications to control a real-time application running on the target PC. You generate real-time applications from Simulink® models.

xPC Target API versus xPC Target COM API

The xPC Target API and xPC Target COM API interfaces provide the same functionality for you to write custom applications. There is no difference in performance or functionality between applications written against either library. Note that the APIs are not threadsafe.

The xPC Target API DLL consists of C functions that you can incorporate into any high-level language application. The xPC Target COM API consists of a suite of interfaces that you can reference while building a graphic user interface (GUI) application. You can incorporate these interfaces using programming environments that work with COM objects. A user can use an application written through either interface to load, run, and monitor an xPC Target application without interacting with MATLAB®. With the xPC Target API, you write the application in a high-level language (such as C, C++, or Java) that works with an xPC Target application; this option requires that you are an experienced programmer. With xPC Target COM API, you use a graphical development environment to create a GUI that works with an xPC Target application. Designed to work with Microsoft COM, the xPC Target COM API conforms to the component object model standard established by Microsoft.

The xPC Target API is distributed with two dynamic link libraries (DLLs) that make it easier to integrate with various development tools, tailoring the development environment to your needs:

- A function library (`xpcapi.dll`)
- A component library (`xpcapicom.dll`)

The following sections describe each library:

- "What Is xPC Target API?" on page 1-4
- "What Is xPC Target COM API?" on page 1-6

Note In this book, second-person references apply to those who write the xPC Target API and COM API applications. For example, “You can assign multiple labels to one tag.” Third-person references apply to those who run the xPC Target API and COM API applications. For example, “You can later distribute this executable to users, who can then use the GUI application to work with target applications.”

What Is xPC Target API?

The xPC Target API consists of a series of C functions that you can call from a C or C++ application. These functions enable you to

- Establish communication between the host PC and the target PC via an Ethernet or serial connection
- Load the target application, a `.dlm` file, to the target PC
- Run that application on the target PC
- Monitor the behavior of the target application on the target PC
- Stop that application on the target PC
- Unload the target application from the target PC
- Close the connection to the target PC

The `xpcapi.dll` file contains the xPC Target API dynamic link library. It contains over 90 functions that enable run-time linking rather than static linking at compile time. The functions provide all the information and accessibility needed to access the target application. Accessing the xPC Target API DLL is beneficial when you are building applications using development environments such as Microsoft Foundation Class Library/Active Template Library (MFC/ATL), DLL, Win32 (non-MFS) program and DLL, and console programs integrating with third-party product APIs (for example, Altia).

All custom xPC Target API applications must link with the `xpcapi.dll` file (xPC API DLL). Also associated with the dynamic link library is the `xpcinitfree.c` file. This file contains functions that load and unload the xPC Target API. You must build this file along with the custom xPC Target API application.

The documentation reflects the fact that the API is written in the C programming language. However, the API functions are usable from other languages and applications, such as C++ and Java.

Note To write a non-C application that calls functions in the xPC Target API library, refer to the compiler documentation for a description of how to access functions from a library DLL. You must follow these directions to access the xPC Target API DLL.

The following chapters describe the xPC Target API in more detail:

- Chapter 2, "xPC Target API" describes how to create a C xPC Target API application.
- Chapter 5, "API Functions and Methods — By Category" and Chapter 6, "API Functions and Methods — Alphabetical List" describe the xPC Target C and COM API functions.

What Is xPC Target COM API?

The xPC Target COM API is an open environment application program interface designed to work with Microsoft COM and the xPC Target API. The xPC Target COM API provides the same functionality as the xPC Target API. It is a programming layer that sits between you and the xPC Target API. The difference is that while the xPC Target API is a dynamic link library of C functions, the xPC Target COM API dynamic link library is an organized collection of objects, classes, and functions. You access this collection through a graphical development environment such as Microsoft Visual Basic. Using such a graphical development environment, you can create a custom GUI application that can work with one xPC Target application. While the xPC Target API requires you to be an accomplished C or C++ programmer, the xPC Target COM API makes no such demand.

The xPC Target COM API library depends on `xpcapi.dll`, the xPC Target dynamic link library. However, the xPC Target API is independent of the xPC Target COM API.

The xPC Target COM API has the following features:

- A DLL component server library — `xpcapicom.dll` is a component server DLL library COM interface consisting of component interfaces that access the target PC. The COM API library enhances the built-in functionality of a programming language by allowing you to easily access the xPC Target API for rapid development of xPC Target GUI.
- Built on top of the xPC Target API — Via an application such as Visual Basic, `xpcapicom.dll`, using a structured object model hierarchy, provides full access to all the data and methods needed to interface with an xPC Target application. It also enables search functionality and bidirectional browsing capabilities. Generally, you view object models by selecting a type and viewing its members. Using the xPC Target COM API library, you can select a member and view the types to which it belongs.
- Programming language independent — This section describes how to create an xPC Target COM API application using Visual Basic. However, the xPC Target COM API interface is not limited to this third-party product. You can add the COM API library to any development environment that can access COM libraries, such as Visual C++ or Java, as well as scripting languages such as Perl, Python, and Basic.

- Ideal for use with Visual Basic — The xPC Target COM API works well with Visual Basic, and extends the event-driven programming environment of Visual Basic.

See Chapter 3, "xPC Target COM API" for a description of how to use the xPC Target COM API library.

Required Products

Refer to "Required Products" in Getting Started with xPC Target documentation for a list of the required xPC Target products. In addition, you need the following products:

- Third-Party Compiler — Use a third-party compiler to build a custom application that calls functions from the xPC API library. Although the xPC API library is written in C, you can write the application that calls these functions in another high-level language, such as C++. You can use any compiler that can generate code for Win32 systems.

  To write a non-C application that calls functions in the xPC Target API library, refer to the compiler documentation for a description of how to access functions from a library DLL. You must follow these directions to access the xPC Target API DLL.

- Third-Party Graphical Development Environment — Use a third-party graphical development environment to build a custom application that references interfaces in the xPC COM API library. Layered on top of the xPC API library, the xPC COM API library enables you to write custom applications using a component object model library. You can use any compiler that can work with component object model (COM) objects.

2

xPC Target API

This chapter describes how to write a custom application using the xPC Target API. This API enables you to write high-level language applications to load an xPC Target application, and run and control it.

Before You Start

Before you start, read this section for important notes on writing custom applications based on the xPC Target API. It is assumed that you already know how to write C or C++ code.

This chapter provides tutorials on how to generate a C application for xPC Target. It also provides some guidelines on using the xPC Target API. Refer to “Visual C Example” on page 2-4 for tutorials that you can follow to create, build, and run a sample Visual C program.

For the xPC Target API function synopses and descriptions, refer to “API Functions and Methods — By Category”.

Important Guidelines

This section describes some guidelines you should keep in mind before beginning to write xPC Target API applications with the xPC Target API DLL:

- You must carefully match the data types of the functions documented in the API function reference. For C, the API includes a header file that matches the data types.
- To write a non-C application that calls functions in the xPC Target API library, refer to the compiler documentation for a description of how to access functions from a library DLL. You must follow these directions to access the xPC Target API DLL
- If you want to rebuild the model `sf_car_xpc.mdl`, or otherwise use MATLAB, you must have xPC Target Version 2.0 or later. This is the version of xPC Target that comes with Release 13 (MATLAB 6.5) or later.

  To determine the version of xPC Target you are currently using, at the MATLAB command line, type

  ```
  xpclib
  ```

  This opens the xPC Target Simulink blocks library. The version of xPC Target should be at the bottom of the window.
- You can work with xPC Target applications with either MATLAB or an xPC Target API application. If you are working with an xPC Target application

simultaneously with a MATLAB session interacting with the target, keep in mind that only one application can access the target PC at a time. To move from the MATLAB session to your application, in the MATLAB Command Window, type

```
close(xpc)
```

This frees the connection to the target PC for use by your xPC Target API application. Conversely, you will need to quit your application, or do the equivalent of calling the function `xPCClosePort`, to access the target from a MATLAB session.

There are a few things that are not covered in Chapter 5, “API Functions and Methods — By Category” and Chapter 6, “API Functions and Methods — Alphabetical List” for the individual functions, because they are common to almost all the functions in the xPC Target API. These are

- Almost every function (except `xPCOpenSerialPort`, `xPCOpenTcpIpPort`, `xPCGetLastError`, and `xPCErrorMsg`) has as one of its parameters the integer variable *`port`*. This variable is returned by `xPCOpenSerialPort` and `xPCOpenTcpIpPort`, and is the placeholder for the communications link with the target PC. The returned value from these two functions should be used in the other functions to ensure that the proper communications channel is used.
- Almost every function (except `xPCGetLastError` and `xPCErrorMsg`) sets a global error value in case of error. The application obtains this value by calling the function `xPCGetLastError`, and retrieves a descriptive string about the error by using the function `xPCErrorMsg`. Although the actual values of the error numbers are subject to change, a zero value always means that the operation completed without errors, while a nonzero value typically signifies an error condition. Note also that the library resets the error value every time an API function is called; therefore, your application should check the error status as soon as possible after a function call.

  Some functions also use their return values (if applicable) to signify that an error has occurred. In these cases as well, you can obtain the exact error with `xPCGetLastError`.

Visual C Example

This release includes an example using the xPC Target API to create a Win32 console application written in C. You can use this example as a template to write your own application.

Before you start, you should have an existing xPC Target application that you want to load and run on a target PC. The following tutorials use the target application `sf_car_xpc.dlm`, built from the Simulink model `sf_car_xpc.mdl`, which models an automatic transmission control system. The automatic transmission control system consists of modules that represent the engine, transmission, and vehicle, with an additional logic block to control the transmission ratio. User inputs to the model are in the form of throttle (%) and brake torque (pound-foot). You can control the target application through MATLAB with the Simulink External Model interface, or through a custom xPC Target API application, which you can create using the tutorials in this chapter.

The topics in this section are

Directories and Files

This directory contains the C source of a Win32 console application that serves as an example for using the xPC Target API. The necessary `sf_car_xpc` files are in the directory

```
C:\matlabroot\toolbox\rtw\targets\xpc\api\VisualC
```

Filename	Description
sf_car_xpc.mdl	Simulink model for use with xPC Target
sf_car_xpc.dlm	Target application compiled from Simulink model
sf_car_xpc.dsp	Project file for API application
sf_car_xpc.c	Source code for API application
sf_car_xpc.exe	Compiled API application
xpcapi.dll	xPC Target API functions for all programming languages

The necessary xPC Target API files are in the directory

```
C:\matlabroot\toolbox\rtw\targets\xpc\api
```

You will need the files listed below for creating your own API application with Microsoft Visual C++.

Filename	Description
xpcapi.h	Mapping of data types between xPC Target API and Visual C
xpcapiconst.h	Symbolic constants for using scope, communication, and data-logging functions
xpcinitfree.c	C functions to upload API from xpcapi.dll
xpcapi.dll	xPC Target API functions for all programming languages

Building the xPC Target Application

The tutorials in this chapter use the prebuilt xPC Target application

```
C:\matlabroot\toolbox\rtw\targets\
xpc\api\VisualC\sf_car_xpc.dlm
```

You can rebuild this application for your example:

1 Create a new directory under your MathWorks directory. For example,

```
D:\mwd\sf_car_xpc2
```

2 Create a Simulink model and save to this directory. For example,

```
sf_car_xpc2.mdl
```

3 Build the target application with Real-Time Workshop® and Microsoft Visual C++. The target application file `sf_car_xpc2.dlm` is created.

Using Another C/C++ Compiler

The tutorials in this chapter describe how to create and build C applications using Microsoft Visual C++. However, to build an xPC Target API C application, you can use any C/C++ compiler capable of generating a Win32 application. You will need to link and compile the xPC Target API application along with `xpcinitfree.c` to generate the executable. The file `xpcinitfree.c` contains the definitions for the files in the xPC Target API and is located at

```
C:\matlabroot\toolbox\rtw\targets\xpc\api
```

Creating a Visual C Application

This tutorial describes how to create a Visual C application. It is assumed that you know how to write C applications. Of particular note when writing xPC Target API applications,

- Call the function `xPCInitAPI` at the start of the application to load the functions.
- Call the function `xPCFreeAPI` at the end of the application to free the memory allocated to the functions.

To create a C application with a program such as Microsoft Visual C++,

1 From the previous tutorial, change directory to the new directory. This is your working directory. For example,

```
D:\mwd\sf_car_xpc2
```

2 Copy the files `xpcapi.h`, `xpcapi.dll`, `xpcapiconst.h`, and `xpcintfree.c` to the working directory. For example,

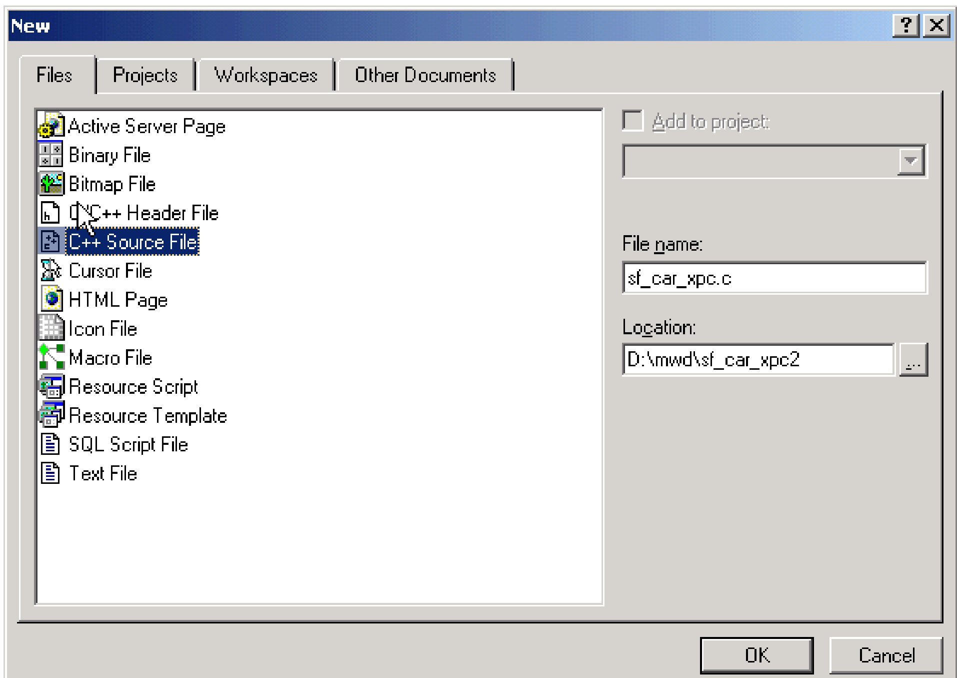
```
D:\mwd\sf_car_xpc2
```

3 Click the **Start** button, choose the **Programs** option, and choose the **Microsoft Visual C++** entry. Select the **Microsoft Visual C++** option.

The Microsoft Visual C++ application is displayed.

4 From the **File** menu, click **New**.

5 At the New dialog, click the **File** tab.



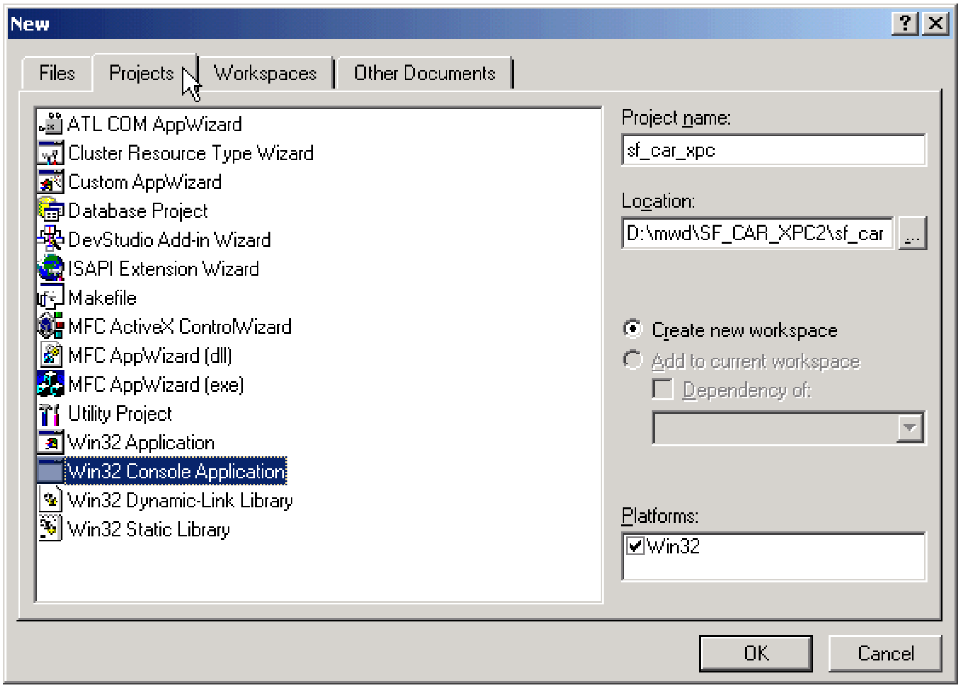
6 In the left pane, select **C++ Source File**. In the right, enter the name of the file. For example, `sf_car_xpc.c`. Select the directory. For example, `C:\mwd\sf_car_xpc2`.

7 Click **OK** to create this file.

8 Enter your code in this file. For example, you can enter the contents of `sf_xpc_car.c` into this file.

9 From the **File** menu, click **New**.

10 At the New dialog, click the **Projects** tab.



11 In the left pane, select **Win32 Console Application**. On the right, enter the name of the project. For example, `sf_car_xpc`. Select the working directory from step 1. For example, `C:\mwd\sf_car_xpc2`.

12 To create the project, click **OK**.

A Win32 Console Application dialog is displayed.

13 To create an empty project, select **An empty project**.

14 Click **Finish**.

15 To confirm the creation of an empty project, click **OK** at the following dialog.

16 To add the C file you created in step 7, from the **Project** menu, select the **Add to Project** option and select **Files**.

17 Browse for the C file you created in step 7. For example,

```
D:\mwd\sf_car_xpc2\sf_car_xpc.c
```

Click **OK**.

18 Browse for the `xpcinitfree.c` file. For example, `D:\mwd\xpcinitfree.c`. Click **OK**.

Note The code for linking in the functions in `xpcapi.dll` is in the file `xpcinitfree.c`. You must compile and link `xpcinitfree.c` along with your custom application for `xpcapi.dll` to be properly loaded.

---

19 If you did not copy the files `xpcapi.h`, `xpcapi.dll`, and `xpcapiconst.h` into the working or project directory, you should either copy them now, or also add these files to the project.

20 From the **File** menu, click **Save Workspace**.

When you are ready to build your C application, go to "Building a Visual C Application" on page 2-10.

Placing the Target Application File in a Different Directory

The `sf_car_xpc.c` file assumes that the xPC Target application file `sf_car_xpc.dlm` is in the same directory as `sf_car_xpc.c`. If you move that target application file (`sf_car_xpc.dlm`) to a new location, change the path to this file in the API application (`sf_car_xpc.c`) and recompile the API application. The relevant line in `sf_car_xpc.c` is in the function `main()`, and looks like this:

```
xPCLoadApp(port, ".", "sf_car_xpc"); checkError("LoadApp: ");
```

The second argument (".") in the call to xPCLoadApp is the path to sf_car_xpc.dlm. The "." indicates that the files sf_car_xpc.dlm and sf_car_xpc.c are in the same directory. If you move the target application, enter its new path and rebuild the xPC Target API application.

Building a Visual C Application

This tutorial describes how to build the Visual C application from the previous tutorial, or to rebuild the example executable sf_car_xpc.exe, with Microsoft Visual C++:

1. To build your own application using the xPC Target API, ensure that the files xpcapi.h, xpcapi.dll, xpcapiconst.h, and xpcinitfree.c are in the working or project directory.

2. If Microsoft Visual C++ is not already running, click the **Start** button, choose the **Programs** option, and choose the **Microsoft Visual C++** entry. Select the **Microsoft Visual C++** option.

3. From the **File** menu, click **Open**.

   The Open dialog is displayed.

4. Use the browser to select the project file for the application you want to build. For example, sf_car_xpc.dsp.

5. If a corresponding workspace file (for example, sf_car_xpc.dsw) exists for that project, a dialog prompts you to open that workspace instead. Click **OK**.

6. Build the application for the project. From the **Build** menu, select either the **Build** project_name.exe or **Rebuild All** option.

Microsoft Visual C++ creates a file named project_name.exe, where project_name is the name of the project.

When you are ready to run your Visual C Application, go to “Running a Visual C xPC Target API Application” on page 2-11.

Running a Visual C xPC Target API Application

Before starting the API application `sf_car_xpc.exe`, ensure the following:

- The file `xpcapi.dll` must either be in the same directory as the xPC Target API application executable, or it must be in the Windows system directory (typically `C:\windows\system` or `C:\winnt\system32`) for global access. The xPC Target API application depends on this file, and will not run if the file is not found. The same is true for other applications you write using xPC Target API functions.
- The compiled target application `sf_car_xpc.dlm` must be in the same directory as the xPC Target API executable. Do not move this file out of this directory. Moving the file requires you to change the path to the target application in the API application and recompile, as described in “Building a Visual C Application” on page 2-10.

Using the xPC Target API C Application

Any xPC Target API application requires you to have a working target PC running at least xPC Target Version 2.0 (Release 13).

This tutorial assumes that you are using the xPC Target API application `sf_car_xpc.exe` that comes with xPC Target. In turn, `sf_car_xpc.exe` expects that the xPC Target application is `sf_car_xpc.dlm`.

If you are going to run a version of `sf_car_xpc.exe` that you compiled yourself using the `sf_car_xpc.c` code that comes with xPC Target, you can run that application instead. Ensure that the following files are in the same directory:

- `sf_car_xpc.exe`, the xPC Target API executable
- `sf_car_xpc.dlm`, the xPC Target application to be loaded to the target PC
- `xpcapi.dll`, the xPC Target API dynamic link library

  If you copy this file to the Windows system directory, you do not need to provide this file in the same directory.

How to Run the sf_car_xpc Executable

1. Create an xPC Target boot disk with a serial or network communication. If you use serial communications, set the baud rate to 115200. Otherwise, create the boot disk as directed in xPC Target Getting Started.

2. Start the target PC with the xPC Target boot disk.

   The target PC displays messages like the following in the top rightmost message area.

   ```
   System:  Host-Target Interface is RS232 (COM1/2)
   ```

   or

   ```
   System:  Host-Target Interface is TCP/IP (Ethernet)
   ```

3. If you have downloaded target applications to the target PC through MATLAB, in the MATLAB window, type

   ```
   close(xpc)
   ```

   This command disconnects MATLAB from the target PC and leaves the target PC ready to connect to another client.

4. On the host PC, open a DOS window. Change directory to

   ```
   C:\matlabroot\toolbox\rtw\targets\xpc\api\VisualC
   ```

   If you are running your own version of `sf_car_xpc.exe`, change to the directory that contains the executable and xPC Target application. For example,

   ```
   D:\mwd\sf_car_xpc2
   ```

5. From that DOS window, enter the command to start the demo application on the host PC and download the target application to the target PC.

   The syntax for the demo command is

   ```
   sf_car_xpc {-t IpAddress:IpPort|-c COMport}
   ```

If you set up the xPC Target boot disk to use TCP/IP, then give the target PC's IP address and IP port as arguments to `sf_car_xpc`, along with the option `-t`. For example, at the DOS prompt, type

```
sf_car_xpc -t 192.168.0.1:22222
```

If you set up the xPC Target boot disk to use RS-232, give the serial port number as a command-line option. Note that indexing of serial ports starts from 0 instead of 1. For example, if you are using serial communication from COM port 1 on the host PC, type

```
sf_car_xpc -c 0
```

On the host PC, the demo application displays the following message:

```
*-------------------------------------------------------------*
*          xPC Target API Demo: sf_car_xpc.                   *
*                                                             *
* Copyright (c) 2000 The MathWorks, Inc. All Rights Reserved. *
*-------------------------------------------------------------*
Application sf_car_xpc loaded. SampleTime 0.001 StopTime: -1
R  Br   Th G  VehSpeed   VehRPM
- ---- -- - --------- ---------
N    0  0 0    0.000    1000.000
```

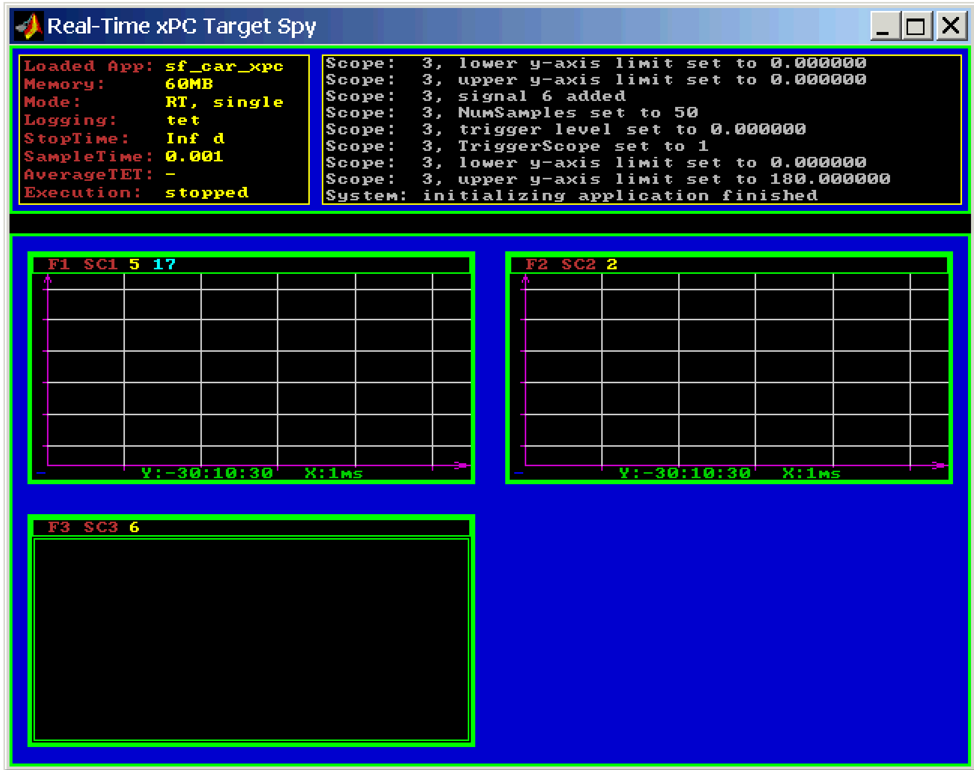
The relevant line here is the last one, which displays the status of the application. The headings are as follows:

R	The status of the target application: R if running, N if stopped
Br	The brake torque; legal values range from 0 to 4000
Th	The throttle as a percentage (0 - 100) of the total
G	Gear the vehicle is in (ranges between 1 and 4)
VehSpeed	Speed of the vehicle in miles per hour
VehRPM	Revolutions per minute of the vehicle engine (0 to 6000)

From this screen, various keystrokes control the target application. The following list summarizes these keys:

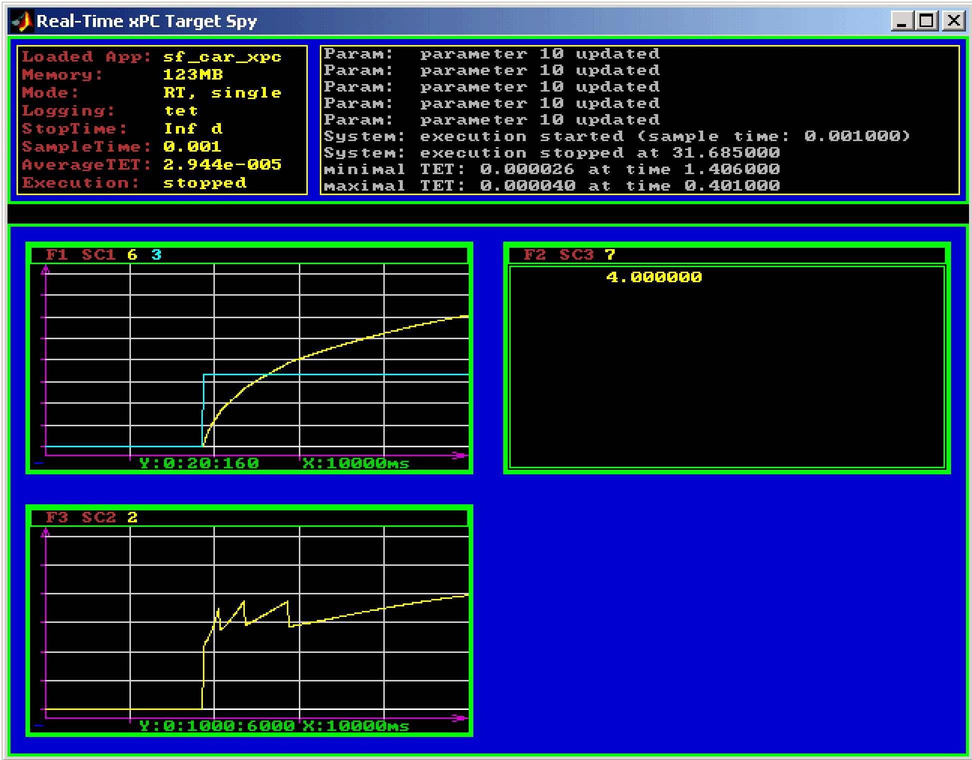
Key	Action
s	Start or stop the application, as appropriate.
T	Increase the throttle by 1 (does not go above 100).
t	Decrease the throttle by 1 (does not go below 0).
B	Increase the brake value by 20 (does not go above 4000). Note that a positive value for the brake automatically sets the throttle value to 0, and a positive value for the throttle automatically sets the brake value to 0.
b	Decrease the brake value by 20 (does not go below 0).
Q or Ctrl+C	Quit the application.

The target PC displays the following messages and three scopes.



6 Hold down the **Shift** key and hold down **T** until the value of `Th` reaches 100.

7 Press **s** to start the application.



The first scope (SC1) shows the throttle rising to a maximum value of 100 and the vehicle speed gradually increasing. The third scope (SC3) shows the vehicle RPM. Notice the changes in the vehicle RPM as the gears shift from first to fourth gear as displayed in the third numerical scope (SC2).

8 When you are done testing the demo application, type **Q** or **Ctrl+C**.

The demo application is disconnected from the target PC, so you can reconnect to MATLAB.

C Code for sf_car_xpc.c

This section contains the C code for the `sf_car_xpc.c` application:

```
/* File:     sf_car_xpc.c
 * Abstract: Demonstrates the use of the xPC Target C-API in Human-Machine
 *           interaction. This file generates a Win32 Console application,
 *           which when invoked loads the sf_car_xpc.dlm compiled application
 *           on to the xPC Target PC.
 *
 *           To build the executable, use the Visual C/C++ project
 *           sf_car_xpc.dsp.
 *
 * Copyright 2000-2004 The MathWorks, Inc.
 */

/* Standard include files */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <ctype.h>
#include <conio.h>
#include <windows.h>

/* xPC Target C-API specific includes */
#include "xpcapi.h"
#include "xpcapiconst.h"

#define SERIAL 0
#define TCPIP  1

/* max and min are defined by some compilers, so we wrap them in #ifndef's */
#ifndef max
#define max(a, b) (((a) > (b)) ? (a) : (b))
#endif
#ifndef min
#define min(a, b) (((a) < (b)) ? (a) : (b))
#endif

/* Global Variables */
int   mode = TCPIP, comPort = 0;
```

```
int   port;
int   thrPID, brakePID, rpmSID, speedSID, gearSID;
char *ipAddress, *ipPort, *pathToApp = NULL;

/* Function prototypes */
double getParam(int parIdx);
void   setParam(int parIdx, double parValue);
void   findParam(char *block, char *param, int *id);
void   findSignal(char *sig, int *id);

void   Usage(void);
void   cleanUp(void);
void   checkError(char *str);
void   processKeys(void);
void   parseArgs(int argc, char *argv[]);
int    str2Int(char *str);


/* Function: main ============================================================
 * Abstract: Main function for the sf_car_xpc demo                          */
int main(int argc, char *argv[]) {
    printf("\n"
           "*-------------------------------------------------------------*\n"
           "*          xPC Target API Demo: sf_car_xpc.                   *\n"
           "*                                                             *\n"
           "* Copyright (c) 2000 The MathWorks, Inc. All Rights Reserved. *\n"
           "*-------------------------------------------------------------*\n"
           "\n");

    parseArgs(argc, argv);
    atexit(cleanUp);
    /* Initialize the API */
    if (xPCInitAPI()) {
        fprintf(stderr, "Could not load api\n");
        return -1;
    }

    if (mode == SERIAL)
        port = xPCOpenSerialPort(comPort, 0);
    else if (mode == TCPIP)
```

```
        port = xPCOpenTcpIpPort(ipAddress, ipPort);
    else {
        fprintf(stderr, "Invalid communication mode\n");
        exit(EXIT_FAILURE);
    }
    checkError("PortOpen: ");

    xPCLoadApp(port, ".", "sf_car_xpc"); checkError("LoadApp: ");
    printf("Application sf_car_xpc loaded, SampleTime: %g  StopTime: %g\n\n",
           xPCGetSampleTime(port), xPCGetStopTime(port));
    checkError(NULL);

    findParam("Throttle", "Value", &thrPID);
    findParam("Brake", "Value", &brakePID);
    findSignal("Engine/rpm", &rpmSID);
    findSignal("Vehicle/mph", &speedSID);
    findSignal("shift_logic/p1", &gearSID);

    processKeys();                      /* Heart of the application */

    if (xPCIsAppRunning(port)) {
        xPCStopApp(port);
    }
    return 0;
} /* end main() */

/* Function: processKeys =====================================================
 * Abstract: This function reads and processes the keystrokes typed by the
 *           user and takes action based on them. This function runs for most
 *           of the program life.                                          */
void processKeys(void) {
    int    c = 0;
    double throttle, brake;

    throttle = getParam(thrPID);
    brake    = getParam(brakePID);
    fputs("\nR    Br    Th  G     VehSpeed     VehRPM  \n", stdout);
    fputs(  "-   ----   --  -    ----------   -------- \n", stdout);
    while (1) {
        if (_kbhit()) {
```

```
    c = _getch();
    switch (c) {
      case 't':
        if (throttle)
            setParam(thrPID, --throttle);
        break;
      case 'T':
        if (brake)
            setParam(brakePID, (brake = 0));
        if (throttle < 100)
            setParam(thrPID, ++throttle);
        break;
      case 'b':
        setParam(brakePID, (brake = max(brake - 200, 0)));
        if (brake)
            setParam(thrPID, (throttle = 0));
        break;
      case 'B':
        if (throttle)
            setParam(thrPID, (throttle = 0));
        setParam(brakePID, (brake = min(brake + 200, 4000)));
        break;
      case 's':
      case 'S':
        if (xPCIsAppRunning(port)) {
            xPCStopApp(port);  checkError(NULL);
        } else {
            xPCStartApp(port); checkError(NULL);
        }
        break;
      case 'q':
      case 'Q':
        return;
        break;
      default:
        fputc(7, stderr);
        break;
    }
} else {
    Sleep(50);
```

```
        }
        printf( "\r%c   %4d  %3d  %1d  %10.3f   %10.3f",
                (xPCIsAppRunning(port) ? 'Y' : 'N'),
                (int)brake, (int)throttle,
                (int)xPCGetSignal(port, gearSID),
                xPCGetSignal(port, speedSID),
                xPCGetSignal(port, rpmSID));
    }
} /* end processKeys() */

/* Function: Usage ==============================================================
 * Abstract: Prints a simple usage message.                                 */
void Usage(void) {
    fprintf(stdout,
            "Usage: sf_car_xpc {-t IPAddress:IpPort|-c num}\n\n"
            "E.g.:  sf_car_xpc -t 192.168.0.1:22222\n"
            "E.g.:  sf_car_xpc -c 1\n\n");
    return;
} /* end Usage() */

/* Function: str2Int ============================================================
 * Abstract: Converts the supplied string str to an integer. Returns INT_MIN
 *           if the string is invalid as an integer (e.g. "123string" is
 *           invalid) or if the string is empty.                            */
int str2Int(char *str) {
    char *tmp;
    int   tmpInt;
    tmpInt = (int)strtol(str, &tmp, 10);
    if (*str == '\0' || (*tmp != '\0')) {
        return INT_MIN;
    }
    return tmpInt;
} /* end str2Int */

/* Function: parseArgs ==========================================================
 * Abstract: Parses the command line arguments and sets the state of variables
 *           based on the arguments.                                        */
void parseArgs(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Insufficient command line arguments.\n\n");
```

```
        Usage();
        exit(EXIT_FAILURE);
    }
    if (strlen(argv[1]) != 2               ||
        strchr("-/",   argv[1][0]) == NULL ||
        strchr("tTcC", argv[1][1]) == NULL) {
        fprintf(stderr, "Unrecognized Argument %s\n\n", argv[1]);
        Usage();
        exit(EXIT_FAILURE);
    }
    mode = tolower(argv[1][1]) == 'c' ? SERIAL : TCPIP;
    if (mode == SERIAL) {
        int tmpInt;
        if ((tmpInt = str2Int(argv[2])) > INT_MIN) {
            comPort = tmpInt;
        } else {
            fprintf(stderr, "Unrecognized argument %s\n", argv[2]);
            Usage();
        }
    } else {
        char *tmp;
        ipAddress = argv[2];
        if ((tmp = strchr(argv[2], ':')) == NULL) {
            /* memory need not be freed as it is allocated only once, will *
             * hang around till app ends.                                  */
            if ((ipPort = malloc(6 * sizeof(char))) == NULL) {
                fprintf(stderr, "Unable to allocate memory");
                exit(EXIT_FAILURE);
            }
            strcpy(ipPort, "22222");
        } else {
            *tmp      = '\0';
            ipPort    = ++tmp;
        }
    }
    return;
} /* end parseArgs() */

/* Function: cleanUp ==========================================================
 * Abstract: Called at program termination to exit in a clean way.          */
```

```
void cleanUp(void) {
    xPCClosePort(port);
    xPCFreeAPI();
    return;
} /* end cleanUp() */

/* Function: checkError =======================================================
 * Abstract: Checks for error by calling xPCGetLastError(); if an error is
 *           found, prints the appropriate error message and exits.         */
void checkError(char *str) {
    char errMsg[80];
    if (xPCGetLastError()) {
        if (str != NULL)
            fputs(str, stderr);
        xPCErrorMsg(xPCGetLastError(), errMsg);
        fputs(errMsg, stderr);
        exit(EXIT_FAILURE);
    }
    return;
} /* end checkError() */

/* Function: findParam ========================================================
 * Abstract: Wrapper function around the xPCGetParamIdx() API call. Also
 *           checks to see if the parameter is not found, and exits in that
 *           case.                                                           */
void findParam(char *block, char *param, int *id) {
    int tmp;
    tmp = xPCGetParamIdx(port, block, param);
    if (xPCGetLastError() || tmp == -1) {
        fprintf(stderr, "Param %s/%s not found\n", block, param);
        exit(EXIT_FAILURE);
    }
    *id = tmp;
    return;
} /* end findParam() */

/* Function: findSignal =======================================================
 * Abstract: Wrapper function around the xPCGetSignalIdx() API call. Also
 *           checks to see if the signal is not found, and exits in that
 *           case.                                                           */
```

```
void findSignal(char *sig, int *id) {
    int tmp;
    tmp = xPCGetSignalIdx(port, sig);
    if (xPCGetLastError() || tmp == -1) {
        fprintf(stderr, "Signal %s not found\n", sig);
        exit(EXIT_FAILURE);
    }
    *id = tmp;
    return;
} /* end findSignal() */

/* Function: getParam ========================================================
 * Abstract: Wrapper function around the xPCGetParam() API call. Also checks
 *           for error, and exits if an error is found.                    */
double getParam(int parIdx) {
    double p;
    xPCGetParam(port, parIdx, &p);
    checkError("GetParam: ");
    return p;
} /* end getParam() */

/* Function: setParam ========================================================
 * Abstract: Wrapper function around the xPCSetParam() API call. Also checks
 *           for error, and exits if an error is found.                    */
void setParam(int parIdx, double parValue) {
    xPCSetParam(port, parIdx, &parValue);
    checkError("SetParam: ");
    return;
} /* end setParam() */

/** EOF sf_car_xpc.c **/
```

xPC Target COM API

This chapter describes how to write a custom application using the xPC Target COM API. This COM API enables you to write COM applications to load, run, and control an xPC Target application.

Before You Start

Before you start, read this section for guidelines on writing custom applications based on the xPC Target COM API. You do not need to be a seasoned C or C++ programmer to follow the procedures in this chapter, or to write custom applications with the xPC Target COM API. You should, however, have some rudimentary programming knowledge.

This chapter provides procedures on how to create xPC Target COM API applications using Microsoft Visual Basic:

- The procedures in this example use the model `xpctank.mdl`. If you want to rebuild this model, or otherwise use MATLAB, you must have xPC Target Version 2.0 or higher.

  To determine which version of xPC Target you are currently using, at the MATLAB command line, type

  ```
  xpclib
  ```

  This opens the xPC Target Simulink blocks library. The version of xPC Target should be at the bottom of the window.

- You can work with xPC Target applications with either MATLAB or an xPC Target COM API application. If you are working with an xPC Target application using an xPC Target COM API application simultaneously with a MATLAB session interacting with the target, keep in mind that only one application can access the target PC at a time. To move from the MATLAB session to your application, in the MATLAB Command Window, type

  ```
  close(xpc)
  ```

  This frees the connection to the target PC for use by your xPC Target COM API application. Conversely, you will need to have your COM API application call the `Close` method to enable access to the target from a MATLAB session.

- Although you are building an xPC Target COM API application, you still need to access the `xpcapi.dll`.

Example Visual Basic GUI Using COM Objects

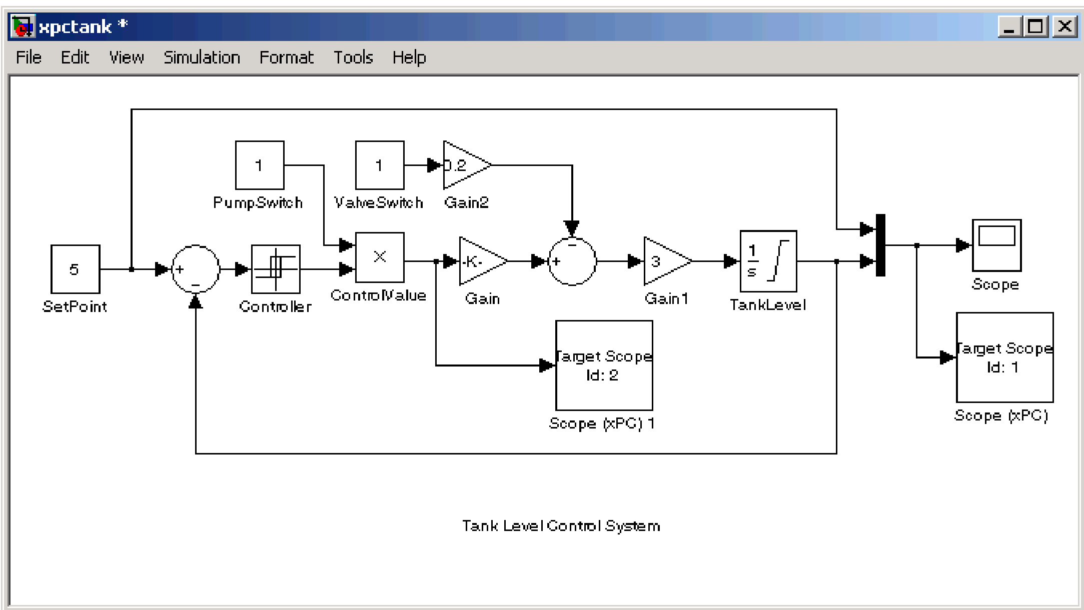
For demonstration purposes this chapter uses the Simulink model `xpctank.mdl` and requests that you enter tags for signals and parameters to create the Simulink model `xpc_tank1.mdl`. You will then build the real-time target application `xpc_tank1.dlm` and the GUI `xpc_tank1_COM.exe` application using the xPC Target COM API library and Microsoft Visual Basic.

This section includes the following topics:

Note This section assumes that you know how to create projects and forms in Microsoft Visual Basic, and that you are familiar with the concept of automatic code completion. For further details on Visual Basic, refer to your Microsoft product documentation.

Description of Simulink Water Tank Model

xPC Target includes the Simulink model `xpctank.mdl`. This is a model of a water tank with a pump, drain, and valve controller.



TankLevel — The water level in the tank is modeled using a limited integrator named TankLevel.

PumpSwitch — The pump can be turned off manually to override the action of the controller. This is done by setting PumpSwitch to 0. When PumpSwitch is 1, the controller can use the control valve to pump water into the tank.

ValveSwitch (drain valve) — The tank has a drain valve that allows water to flow out of the tank. Think of this as water usage or consumption that reduces the water level. This behavior is modeled with the constant block named ValveSwitch, the gain block Gain2, and a summing junction. The minus sign on the summing junction has the effect of producing a negative flow rate (drain), which reduces the water level in the tank.

When ValveSwitch is 0 (closed), the valve is closed and water cannot flow out of the tank. When ValveSwitch is 1 (open), the valve is open and the water level is reduced by draining the tank.

Controller — The controller is very simple. It is a bang-bang controller and can only maintain the selected water level by turning the control valve (pump valve) on or off. A water level set point defines the desired median water level. Hysteresis enables the pump to avoid high-frequency on and off cycling. This is done using symmetric upper and lower bounds that are offsets from the median set point. As a result, the controller turns the control valve (pump valve) on whenever the water level is below the set point minus the offset. The summing junction compares this lower bound against the tank water level to determine whether or not to open the control valve. If the pump is turned on (PumpSwitch is 1) water is pumped into the tank. When the water level reaches or exceeds the set point plus the upper bound, the controller turns off the control valve. When the water level reaches this boundary, water stops pumping into the tank.

Scope blocks — A standard Simulink Scope block is added to the model for you to view signals during a simulation. xPC Target Scope blocks are added to the model for you to view signals while running the target application. Scope id:1displays the actual water level and the selected water level in the tank. Scope id:2 displays the control signals. Both scopes are displayed on the target PC using a scope of type target.

The `xpctank.mdl` model is built entirely from standard Simulink blocks and scope blocks from xPC Target. It does not differ in any way from a model you would normally use with xPC Target.

Creating a Simulink Target Model

A target application model is a Simulink model that describes your physical system 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.

You do not have to modify this model when you use it with Virtual Reality Toolbox or other third-party graphical elements.

Create a target application model before you tag block parameters and block signals to create a custom graphical interface:

1. In the MATLAB Command Window, type

   ```
   xpctank
   ```

   A Simulink model for a water tank opens. This model contains a set of equations that describe the behavior of a water tank and a simple controller.

   The controller regulates the water level in the tank. This model contains only standard Simulink blocks and you use it to create the xPC Target application.

2. From the **File** menu, click **Save as** and enter a new filename. For example, enter `xpc_tank1` and then click **OK**.

Note If you save your own copy of `xpctank`, be sure to be in the directory that contains that model before calling it from the MATLAB window.

Your next task is to mark the block properties and block signals. See "Tagging Block Parameters" on page 3-8 and "Tagging Block Signals" on page 3-11. Building an xPC Target application that has been tagged generates a model-specific COM library, `model_nameifaceCOM.dll`, which you can later reference when writing your xPC Target COM API application.

Tagging Block Parameters

Tagging parameters in your Simulink model enables you to generate a model-specific COM library to provide access to model parameter IDs via the xPC Target COM API library. These interface blocks contain the parameters you connect to control devices (such as sliders) in your model. Tagging parameters makes it easier for you to refer to these parameters later, when you write your xPC Target COM API application.

Note If you do not tag parameters before you generate your Simulink model, you must specify model parameters manually. See "Referencing Parameters and Signals Without Using Tags" on page 3-41 for this procedure.

This procedure uses the model`xpc_tank1.mdl` (or `xpctank.mdl`) as an example. See "Creating a Simulink Target Model" on page 3-7.

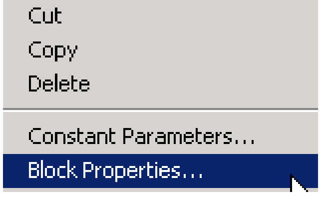
Note The `xpctank` model that comes with xPC Target contains tags from the example for creating custom user interfaces in the xPC Target User's Guide . As you follow the procedures in this section and the section "Tagging Block Signals" on page 3-11, you should remove any existing tags before adding the new tags.

---

1 Open a Simulink model. For example, in the MATLAB window type `xpc_tank1` or `xpctank`.

2 Point to a Simulink block, and then right-click. For example, right-click the SetPoint block.

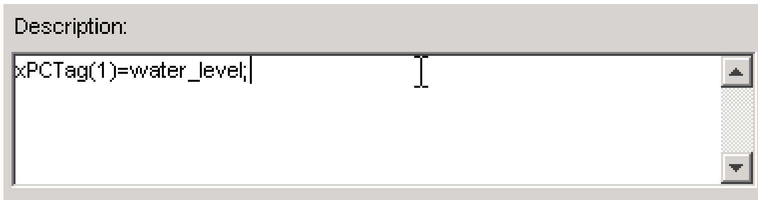
3 From the menu, click **Block Properties**.



A block properties dialog box opens.

4 In the **Description** box, delete the existing tag and 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:

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

- `index_n` — Index of a block parameter. Begin numbering parameters with an index of `1`.
- `label_n` — Name for a block parameter to connect to a property for the parameter you tag in the model. Separate the labels with a space, not a comma.

`label_1...label_n` must consist of the same identifiers as those used by C/C++ to name functions, variables, and so forth. Do not use names like `-foo`.

You can assign multiple labels to one tag, such as

```
xPCTag(1)=label;xPCTag(1)=label2;
```

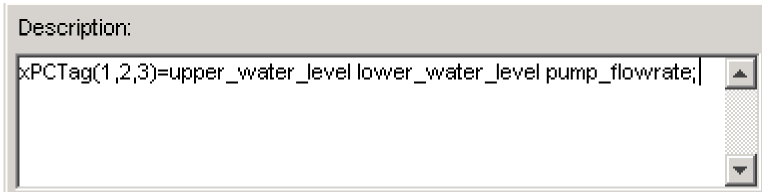
You might want to assign multiple labels if you want to tag a parameter for different purposes. For example, you can tag a parameter to create a model-specific COM library. You might also want to tag a parameter to enable the function `xpcsliface` to generate a user interface template model.

You can also issue one tag definition per line, such as

```
xPCTag(1)=label;
xPCTag(2)=label2;
```

5 Repeat step 4 for the remaining parameters you want to tag.

For example, for the Controller block, enter the tag



For the PumpSwitch and ValveSwitch blocks, enter the tags

To tag a block with four properties, use the following syntax:

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

To tag a block with at least four properties for the second and fourth 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 tag block signals, if you have not already done so; then, create the model. See "Tagging Block Signals" on page 3-11.

Tagging Block Signals

Tagging signals in your Simulink model enables you to generate a model-specific COM library to provide access to model signal IDs via the COM API library. These interface blocks contain the signals you connect to display devices (such as labels) in your model. Tagging signals makes it easier for you to refer to these signals later, when you write your xPC Target COM API application. After you tag signals, you will be ready to build your xPC Target application.

Note If you do not tag signals before you generate your Simulink model, you must specify model signals manually. See "Referencing Parameters and Signals Without Using Tags" on page 3-41 for this procedure.

This procedure uses the model `xpc_tank1.mdl` (or `xpctank.mdl`) as an example. See "Creating a Simulink Target Model" on page 3-7.

Note The `xpctank` model that comes with xPC Target contains tags from the example for creating custom user interfaces in the xPC Target User's Guide. As you follow the procedures in this section and the section "Tagging Block Parameters" on page 3-8, you should remove any existing tags before adding the new tags.

Notice that you cannot select signals on the output ports of any virtual blocks such as Subsystem and Mux blocks. Also, you cannot select signals on any function call signal output ports.

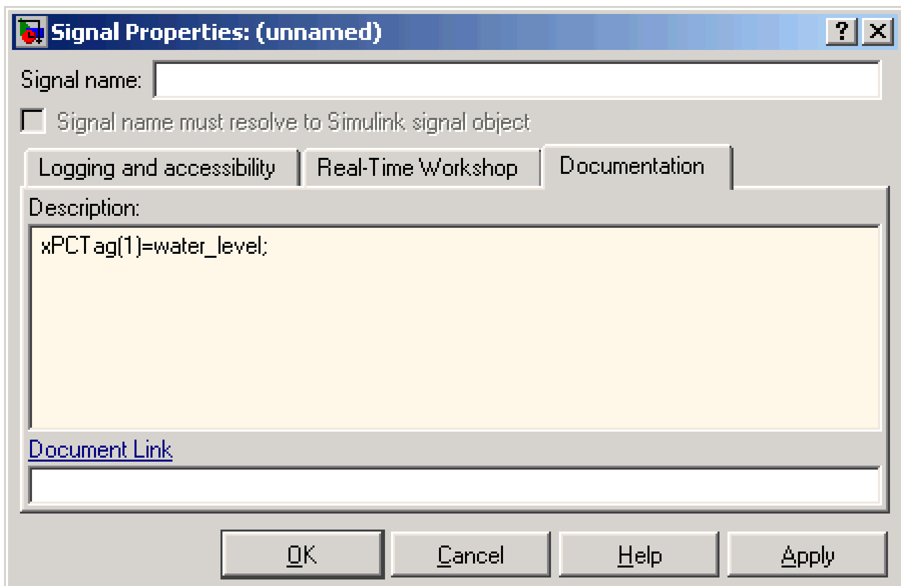
1 Open a Simulink model. For example, in the MATLAB window type `xpc_tank1` or `xpctank`.

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

3 From the menu, click **Signal Properties**. For example, right-click the signal line from the TankLevel block.

A Signal Properties dialog box opens.

4 Select the **Documentation** tab.

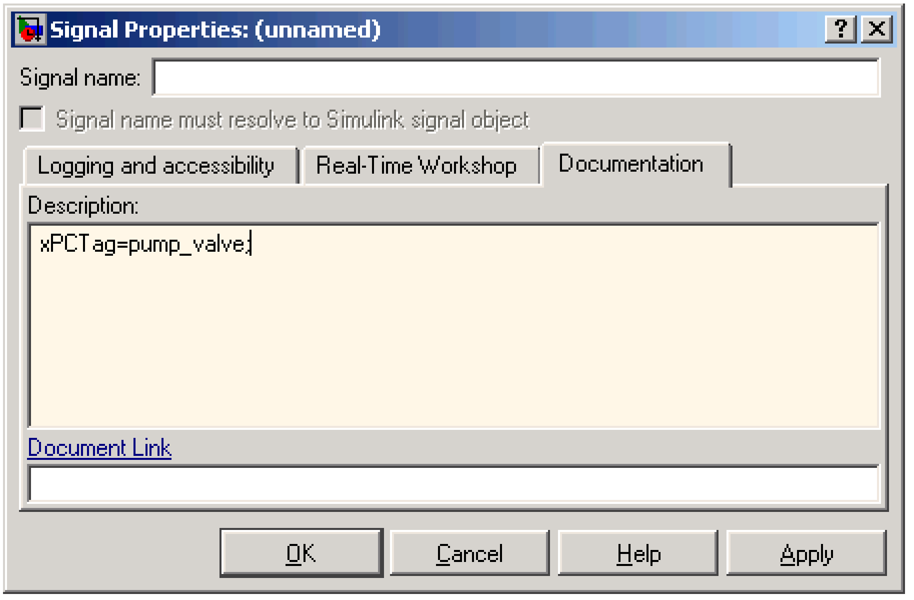


5 In the **Description** box, enter a tag to the signals for this line.

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.

6 Repeat step 5 for the remaining signals you want to tag.

For example, for the signal from the ControlValve block, enter the tag `pump_valve`.



Signal tags have the following syntax:

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

- `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 to connect to a property for the signal you tag in the model. Separate the labels with a space, not a comma.

`label_1...label_n` must consist of the same identifiers as those used by C/C++ to name functions, variables, and so forth. Do not use names like `-foo`.

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

```
XPCTag=label;
```

You can assign multiple labels to one tag, such as

```
xPCTag(1)=label;xPCTag(1)=label2;
```

You might want to assign multiple labels if you want to tag a signal for different purposes. For example, you can tag a signal to create a model-specific COM library. You might also want to tag a signal to enable the function `xpcsliface` to generate a user interface template model.

You can also issue one tag definition per line, such as

```
xPCTag(1)=label;
xPCTag(2)=label2;
```

To tag 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 tag 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;
```

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

   ```
   xpc_tank1
   ```

Create the target application. See "Creating the Target Application and Model-Specific COM Library" on page 3-14.

Creating the Target Application and Model-Specific COM Library

Use this procedure to create a target application that you want to connect to a GUI application and the model-specific COM interface library (`model_nameCOMiface.dll`).

After you copy a Simulink model and tag the block parameters and block signals, you can create a target application and download it to the target PC.

This procedure uses the Simulink model `xpc_tank1.mdl` (or `xpctank.mdl`) as an example (see “Creating a Simulink Target Model” on page 3-7).

1. Start or reset the target PC with an xPC Target boot disk in the floppy drive. Ensure that there is no other application currently loaded on the target PC.

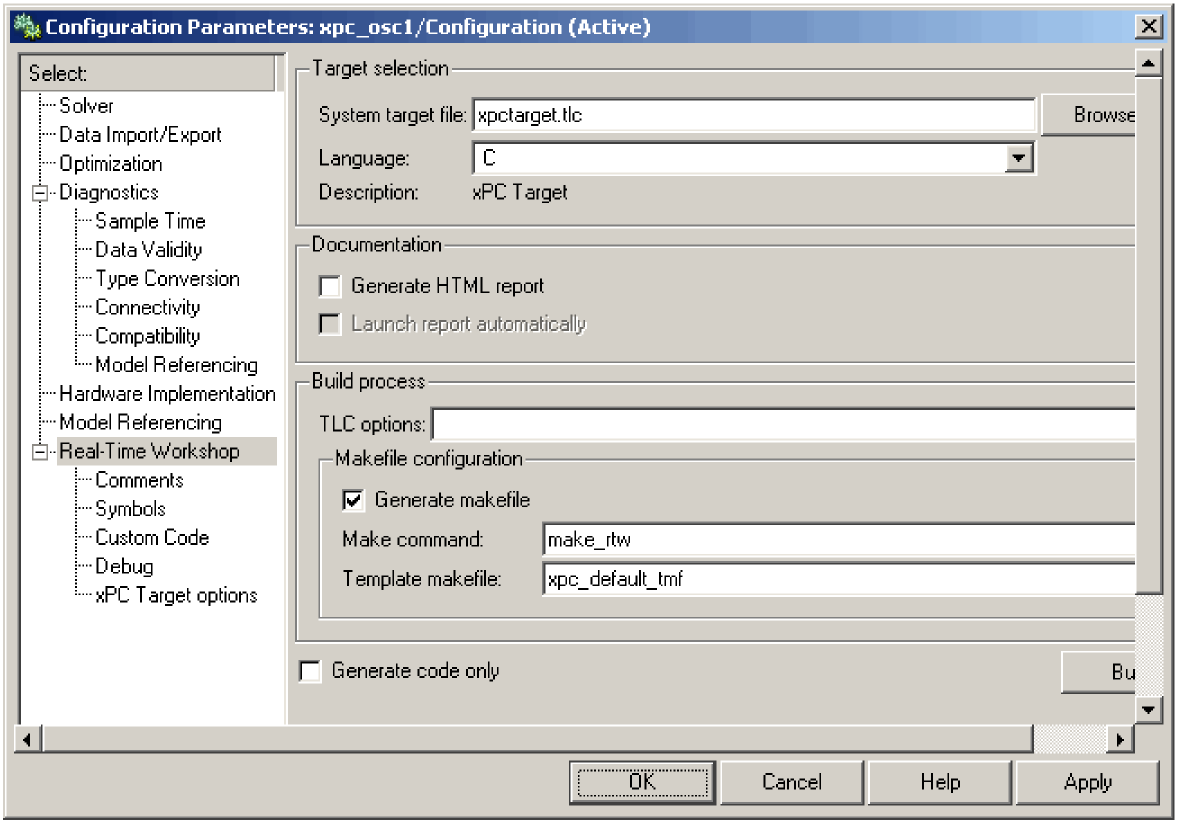
2. If this is a new release of the product, ensure that you have configured the host PC with the appropriate settings, including the compiler.

3. In the MATLAB window, type `xpc_tank1` or `xpctank`.

   A Simulink window opens with the model`.mdl` file.

4. From the **Simulation** menu, click **Configuration Parameters**.

   The Configuration Parameters dialog is displayed for the model.



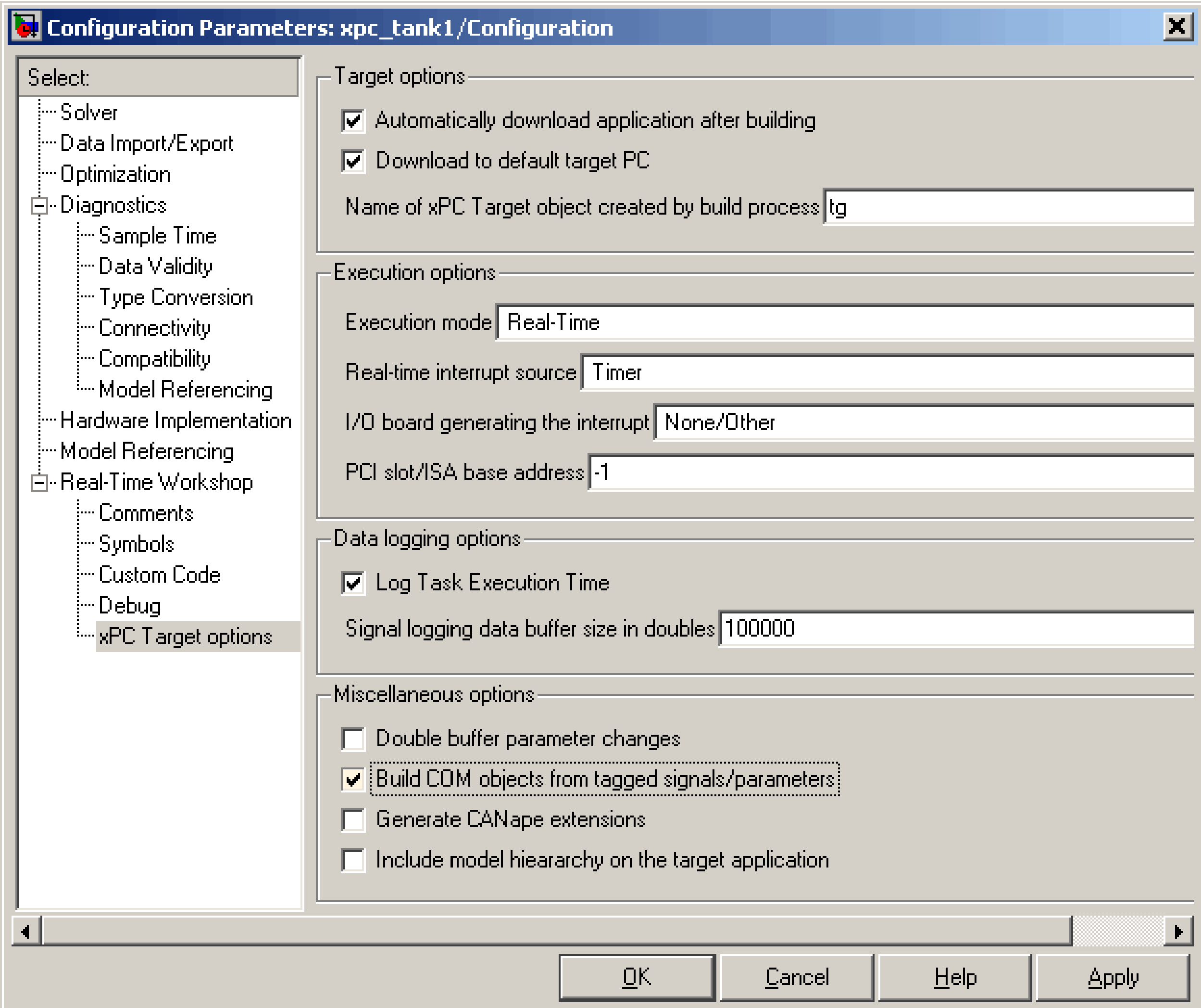
5 In the left pane, click the **Real-Time Workshop** node.

6 In the **Target selection** section, click the **Browse** button at the **RTW system target file** list. Click `xpctarget.tlc` if it is not already selected, then click **OK**.

7 In the left pane, click the **xPC Target options** node.

The **xPC Target options** pane is displayed.

8 Select the **Build COM objects from tagged signals/parameters** check box.



9 Click the **Solver** node.

The **Solver** pane is displayed.

10 Check that the **Stop time** is long enough for you to interact with the target application.

11 Click **OK** to save and exit.

12 From the **Tools** menu, point to **Real-Time Workshop**, and then click **Build model**.

Real-Time Workshop, xPC Target, and a third-party C compiler create the target application `xpc_tank1.dlm` and the COM object library `xpc_tank1COMiface.dll`. The target application is also downloaded to the target PC.

13 If you want, you can close MATLAB.

Note To create the target application and build associated COM objects from the tagged signals and parameters, you must use the Visual C compiler. You cannot use the Watcom compiler to build these COM objects.

Your next task is to create a Visual Basic API application using COM objects. This API application connects and controls the target application. See "Creating a New Visual Basic Project" on page 3-20. For more information about model-specific COM interface library, refer to "Model-Specific COM Interface Library (model_nameCOMiface.dll)" on page 3-18.

Model-Specific COM Interface Library (model_nameCOMiface.dll)

The generated model-specific COM interface library is a DLL component server library that enhances programming using the xPC Target COM API library. A model-specific COM interface library is specific to the model from which it is generated; do not reference a model-specific library for another model. If you choose not to generate a model-specific COM interface library, refer to "Referencing Parameters and Signals Without Using Tags" on page 3-41 for a description of how to otherwise reference parameters and signals in the xPC Target COM API application.

The mode-specific COM interface library allows users easy access to preselected tagged signals and desired tagged parameters for use in conjunction with the xPC Target COM API xPC Target and xPCScope object signal monitoring and parameter member functions such as `xPCTarget.GetParam`, `xPCTarget.SetParam`, and `xPCTarget.GetSignal`.

The xPC Target COM generated objects are of two types:

- `model_namebio`
- `model_namept`

where `model_name` is the name of the Simulink model. The `model_namebio` type is for tagged block I/O signals and the `model_namept` type is for tagged parameters.

Model-Specific COM Signal Object Classes

Model-specific COM signal classes have two types of members in which you are interested, the `Init` function and class properties. You will find these members in the `model_namebio` class, where `model_name` is the name of your model.

The `Init` function invokes the `Init` method once, passing it the `Ref property from` the `xPCProtocol` class. This method initializes the object to communicate with the appropriate target PC to access the signal identifiers when accessing the object's properties. Refer to the call in the Visual Basic code example in "Creating the Load Procedure" on page 3-33.

Each class has a list of properties (specified in the `Tag` syntax in the **Description** field of the signal property). These properties return the xPC Target signal identifiers or signal numbers of the tagged signals. The generated property name is the name specified in the tagged signal description using the following syntax:

```
xPCTag=Property name;
```

For example, in the model `xpc_tank1.mdl`, there are two signal tags in the **Description** field:

- The output from the integrator block labeled TankLevel is tagged `xPCTag=water_level`.
- The output from the multiply block labeled ControlValve is tagged `xPCTag=pump_valve`.

Model-Specific COM Parameter Object Classes

Model-specific COM signal classes have two types of members in which you are interested, the `Init` function and class properties. You will find these members in the `model_namept` class, where `model_name` is the name of your model.

The `Init` function invokes the `Init` method once, passing it as input the `Ref` property from the `xPCProtocol` class. This method initializes the object to communicate with the appropriate target PC to access the parameter identifiers when accessing the object's properties. Refer to the call in the Visual Basic code example in "Creating the Load Procedure" on page 3-33.

Each class has a list of properties (specified in the `Tag` syntax in the **Description** field of the block property). These properties return the xPC Target parameter identifiers of the tagged parameters. The generated property name is the name specified in the tagged signal description using the following syntax:

```
xPCTag(1)=Property name;
```

For example, in the model `xpc_tank1.mdl`, there are two parameter tags in the **Description** field:

- The parameter for SetPoint blocks is tagged `xPCTag=set_water_level`;
- The parameters for the Controller block are tagged `xPCTag(1,2,3,)=upper_water_level lower_water_level pump_flowrate`;

Creating a New Visual Basic Project

The following procedures describe how you can create a Visual Basic project to take advantage of the xPC Target COM API to create a custom GUI for the xPC Target application. The procedures build on the `xpctank` (`xpc_tank1`) model you saved earlier (see "Creating the Target Application and

Model-Specific COM Library" on page 3-14). The Visual Basic environment allows you to interact with your target application using a GUI while the target application is running in real time on the target PC.

The procedures for the following topics apply to Microsoft Visual 6.0. To use Microsoft Visual 7.1 or 8.0 instead, see "Creating a New Visual Basic Project Using Microsoft Visual 7.1 or 8.0" on page 3-47.

1 Create a new project directory.

   From the directory *matlabroot*\toolbox\rtw\targets\xpc\api, copy the file `xpcapi.dll` (API library) to this new project directory. Alternatively, you can copy the file `xpcapi.dll` into the Windows system directory.

   You do not need to copy `xpcapiCOM.dll` (the COM API library) into the current directory, but ensure that it is registered in your system (see "Registering Dependent Dynamic Link Libraries" on page 3-47.)

2 From your MATLAB working directory, copy the files `model_name.dlm` (target application) and `model_nameCOMiface.dll` (model-specific COM library) to the new project directory.

3 While in this project directory, open Visual Basic. From the **File** menu, click **New Project**.

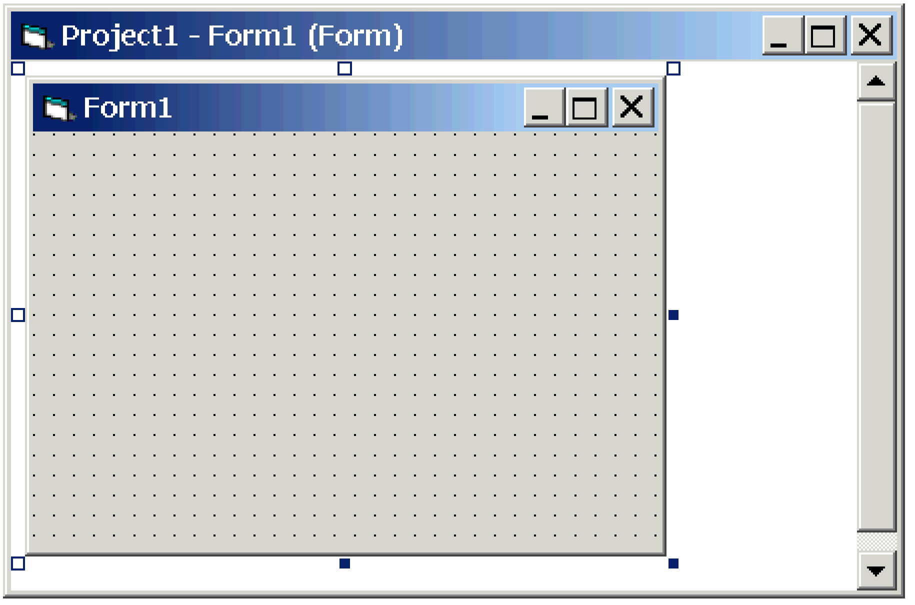
   The New Project dialog box opens.

Note Be sure to open the Visual Basic project from the project directory itself, not from Visual Basic.

---

4 Select **Standard EXE**, and then click **OK**.

The Visual Basic Integrated Development Environment opens with a blank form.



5 From the **File** menu, click **Save Project As** and enter a filename for the form and the project. For example, for the form, enter

```
xpc_tank1_COM.frm
```

At the project prompt, enter

```
xpc_tank1_COM.vbp
```

Referencing the xPC Target COM API and Model-Specific COM Libraries

You need to reference the xPC Target COM API and model-specific COM libraries so that Visual Basic will use them in the current project. Assuming that you created the Visual Basic project as described in the preceding procedure, reference the library as described in this procedure:

1 From the **Project** menu, click **References**.

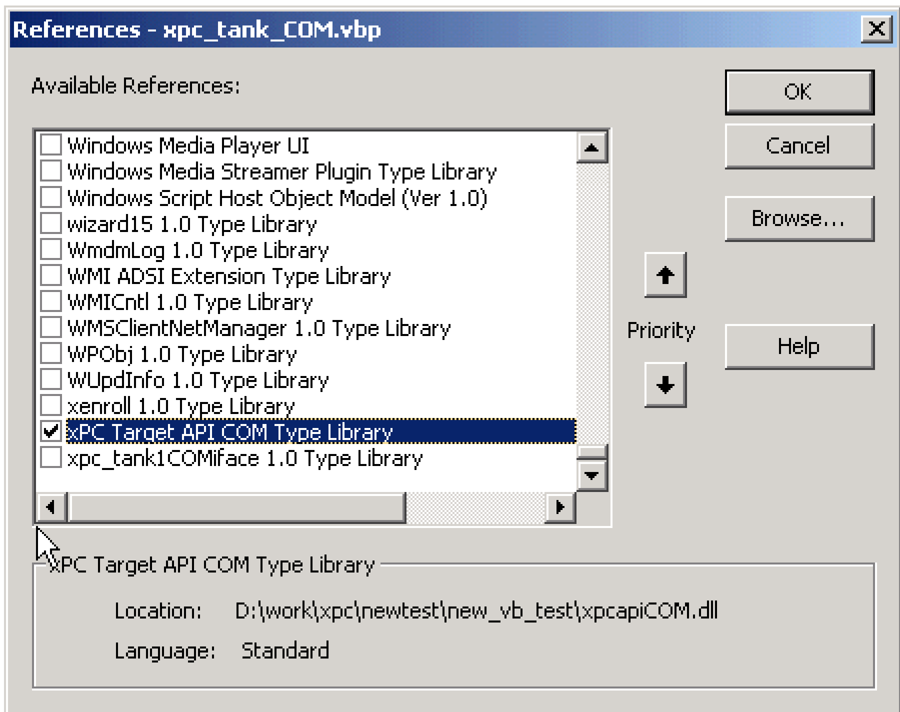
   The References dialog box opens.

2 Select the **COM** tab.

3 Scroll down the **Component Name** list to the bottom. Select the **xPC Target API COM Type Library** check box.

4 Click **Select**.

5 Click **OK**.



The xPC Target COM API Type library (`xpcapiCOM.dll`) is now available for use in your project.

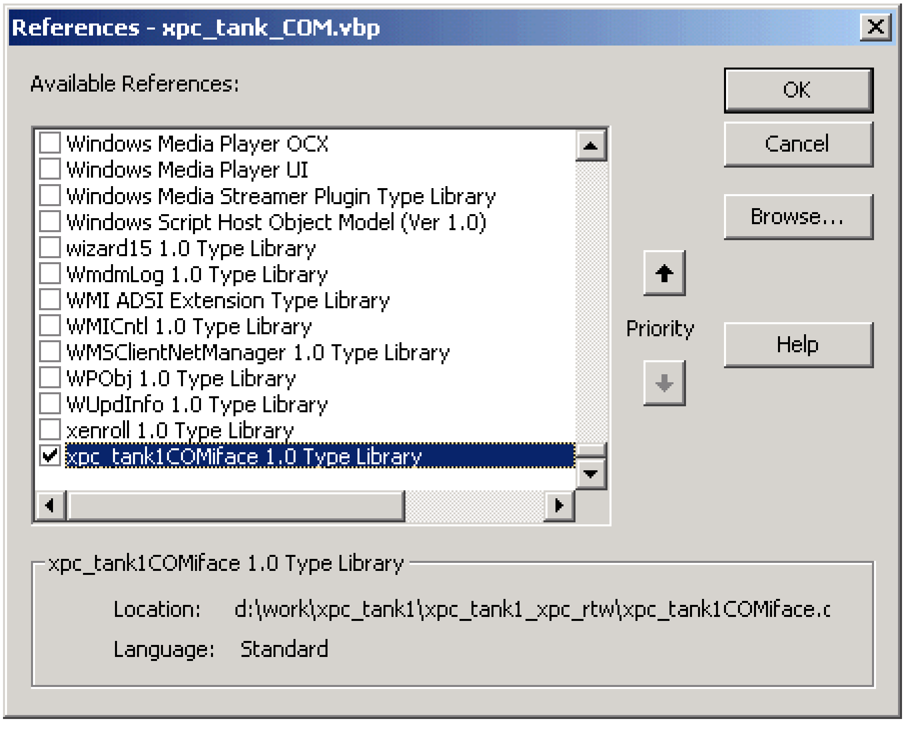
6 To add the model-specific COM library, click **References** again from the **Project** menu.

The References dialog box opens.

7 Scroll to find your model name. Select the check box **xpc_tank1COMiface 1.0 Type Library**.

8 Click **Select**.

9 Click **OK**.



The model-specific COM API Type Library (`xpc_tank1COMiface.dll`) is now available for use in your project. Sections "Viewing Model-Specific COM Signal Object Classes" on page 3-25 and "Viewing Model-Specific COM Parameter Object Classes" on page 3-26 describe how to look at class objects.

Because the xPC Target COM API is an add-on to Visual Basic, it might help to know a bit about Visual Basic before going much farther with using the COM API. The section “Creating the Graphical Interface” on page 3-27 guides you through using Visual Basic to create a project for the `xpctank` or (`xpc_tank1`) model.

Viewing Model-Specific COM Signal Object Classes

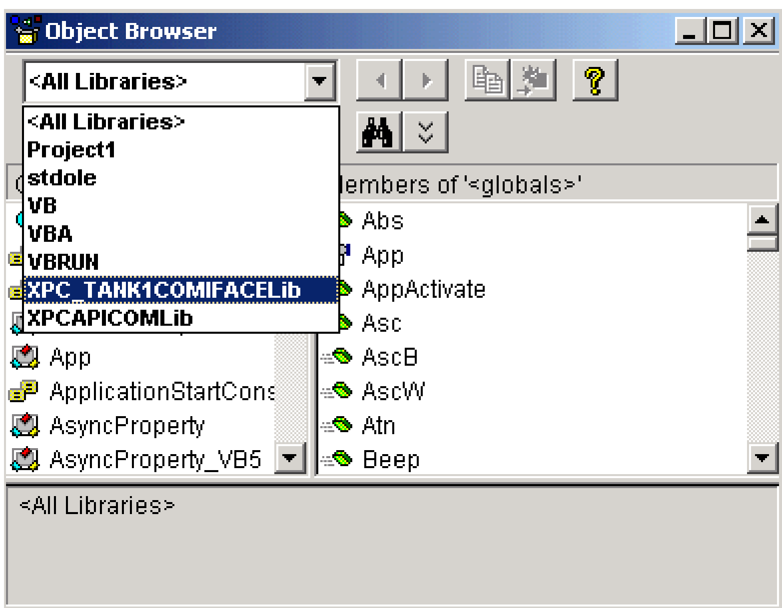
After you create a Visual Basic project and reference the xPC Target COM API and model-specific COM libraries, you can use the Visual Basic Object browser (click the **View** menu and select **Object Browser**) to look at the objects for the `xpctankbio` or `xpc_tank1bio` class:

1. From the **View** menu, select **Object Browser**.

   A dialog box pops up with a drop-down list containing all the type library information for a project.

2. Select the drop-down list for the project/library.

   A list of the project libraries appears.



3 Select model_nameCOMIFACELib.

   The classes in your model appear.

4 To view the objects of a class, select that class.

   The objects in your class appear.

The xpctankbio (or xpc_tank1bio) class contains the function Init and the two properties

- water_level
- pump_valve

Viewing Model-Specific COM Parameter Object Classes

After you create a Visual Basic project and reference the xPC Target COM API and model-specific COM libraries, you can use the Visual Basic Object browser (click the **View** menu and select **Object Browser**) to look at the objects for the xpctankpt or xpc_tank1pt class:

1 From the **View** menu, select **Object Browser**.

   A dialog box pops up with a drop-down list containing all the type library information for a project.

2 Select the drop-down list for the project/library.

   A list of the project libraries appears.

3 Select model_nameCOMIFACELib.

   The classes in your model appear.

4 To view the objects of a class, select that class.

   The objects in your class appear.

The xpctankpt (or xpc_tank1pt) class contains the method Init and the member properties

- pump_switch
- upper_water_level
- lower_water_level
- pump_flowrate
- water_level
- drain_valve

Creating the Graphical Interface

Forms are the foundation for creating the interface of a Visual Basic application. You can use forms to add windows and dialog boxes to your Visual Basic application. You can also use them as containers for items that are not a visible part of the application's interface. For example, you might have a form in your application that holds a timer object.

The first step in building a Visual Basic application is to create the forms that are the basis for your application's interface. Then you create the objects that make up the interface on the forms. This section assumes that you have a Visual Basic project (see "Creating a New Visual Basic Project" on page 3-20). For this first application, you will use four types of controls from the toolbox:

- Button
- Timer
- Label
- Scrollbar

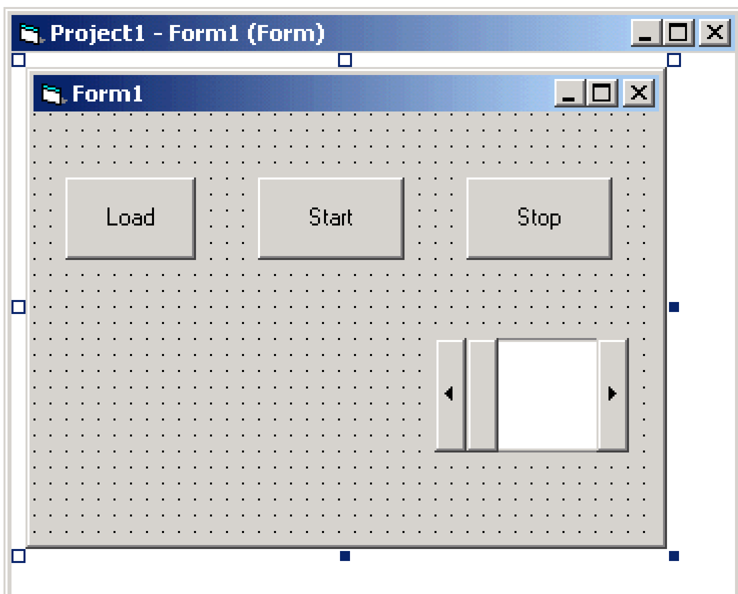
1. Open xpc_tank1_COM.vbp.
2. On the left, from the **General** tool panel, click and drag the **Button** icon  to the form to create a button.
3. Repeat for a second button.
4. If you want to view signal data on the host, return to the **General** tool panel and click and drag the **Timer** icon  to the form to create a timer.

5 If you want to view signal data on the host, add a **Label** control to the form. Return to the **General** tool panel and click and drag the **Label** icon A to the form to create a label.

6 If you want to be able to vary the parameter input to the target, return to the **General** tool panel and click and drag the **HScrollBar** icon to the form.

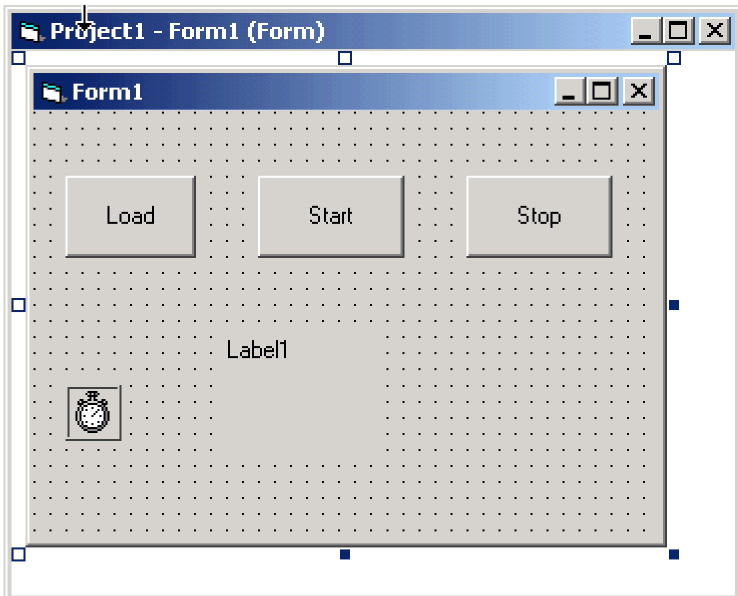
7 Next, name your new form objects. Right-click the first button and select **Properties**. This brings up the Properties dialog box. In the **Caption** box, enter `Load`. Repeat for the second button, but enter `Start`. Repeat for the third button, but enter `Stop`. (If you are unsure about how to work with properties, refer to the procedure "Setting Properties" on page 3-29.) After you name your new form objects and set whatever other parameters you want (for example, if you use a timer you must increase the `Interval` parameter), you can write the code behind these objects using the Visual Basic code editor window (refer to "Writing Code" on page 3-31).

If you added a scroll bar to your project, it should look similar to the figure below.



If you added a timer and label to your project, it should look similar to the figure below.

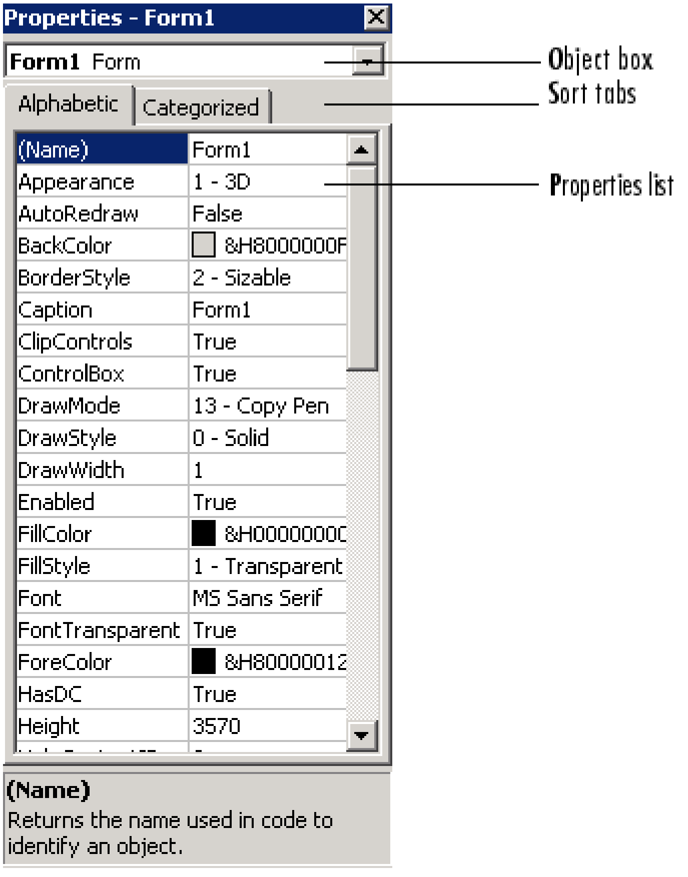
Note If you add a timer, remember to increase the interval of the timer to a value greater than the default value of 0. Right-click the timer and select **Properties**. This brings up the Properties dialog box. In the **Interval** box, enter a value greater than 0, for example, 100.



Setting Properties

This procedure describes how to set properties for the Visual Basic objects you created on your form. If you already know how to set properties for Visual Basic objects, proceed to “Writing Code” on page 3-31.

The **Properties** window in the following figure provides an easy way to set properties for all objects on a form. To open the **Properties** window, choose the **Properties Window** command from the **View** menu, click the **Properties Window** button on the toolbar, or use the context menu for the control.



The **Properties** window consists of the following elements:

- Object box — Displays the name of the object for which you can set properties. Click the arrow to the right of the object box to display the list of objects for the current form.

- Sort tabs — Choose an alphabetic listing of properties or a hierarchical view divided by logical categories, such as those dealing with appearance, fonts, or position.
- Properties list — The left column displays all the properties for the selected object. You can edit and view settings in the right column.

To set properties from the **Properties** window,

1. From the **View** menu, choose **Properties**, or click the **Properties** button on the toolbar.

   The **Properties** window displays the settings for the selected form or control.

2. From the properties list, select the name of a property.

3. In the right column, type or select the new property setting.

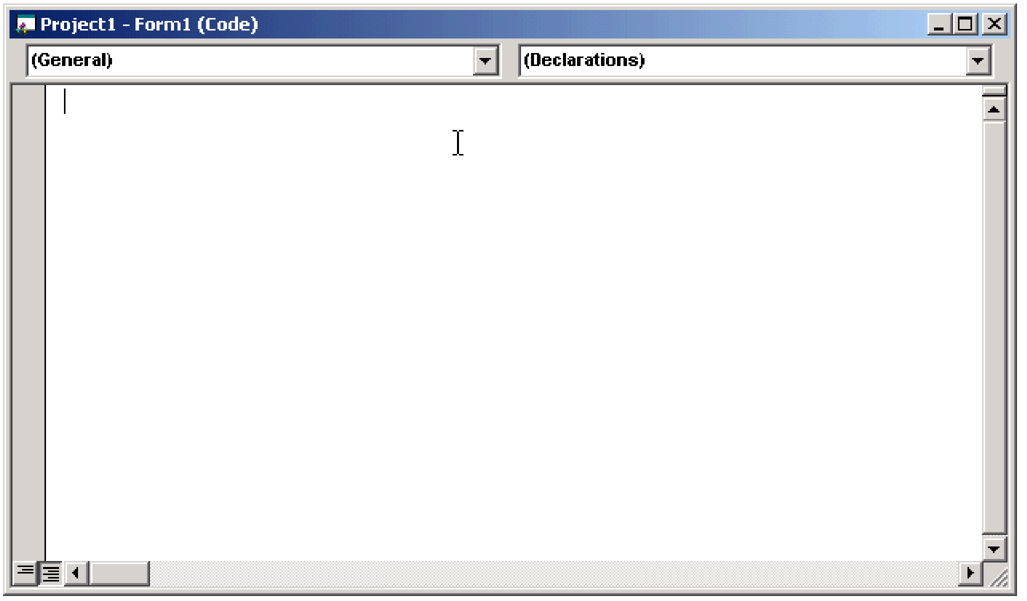
   Enumerated properties have a predefined list of settings. You can display the list by clicking the down arrow at the right of the settings box, or you can cycle through the list by double-clicking a list item.

You can also set object properties directly in the code by using the following dot notation: `Object.propertyname=value.`

Writing Code

The code editor window is where you write Visual Basic code for your application. Code consists of language statements, constants, and declarations. Using the code editor window, you can quickly view and edit any of the code in your application.

The code editor window has three panes. The top leftmost pane is the object list box. It is a dropdown list that contains all the form controls in your project, plus a general section for generic declarations. The top rightmost pane contains a procedure list box. For the selected or active control in the object list box, the procedure list box displays the available procedures, or events. Visual Basic predefines the possible procedures. The third pane contains the code for the Visual Basic application. See the following figure for a sample code editor window.



In the general declarations section, declare a reference to the xPC Target COM objects that you are using to interface with the xPC Target objects. The following are the objects you need to declare:

- `xPCProtocol` — Reference the classes corresponding to the target PC running the target application and initialize the xPC Target API dynamic link library. At a minimum, you must declare this object.
- `xPCTarget` — Reference the classes for interfacing with the target application. At a minimum, you must declare this object.
- `xPCScope` — If the API application requires signal data, reference the class for interfacing with xPC Target scopes. You need to declare a scope if you want to acquire data from scopes or display data on scopes.
- `model_namept` — This is the COM object for tunable model/application parameters.
- `model_namebio` — This is the COM object for model/target application signals.

Creating the General Declarations

This procedure describes how to create the general object declarations for the `xpctank` (or `xpc_tank1`) model:

1 Double-click the form or, from the **View** menu, select **Code**.

   The code editor window box opens for the control.

2 Select the General object.

3 Select **Declarations** in the procedure list box.

   A *template* for the declarations procedure is now displayed in the code editor window.

4 Enter declarations for the xPC Target COM objects you are using.

```
Public protocol_obj As xPCProtocol
Public target_obj As xPCTarget
Public scope_obj As xPCScopes
```

5 Enter declarations for the model-specific COM objects you are using.

```
Public parameters_obj As xpc_tank1pt
Public signals_obj As xpc_tank1bio
```

Creating the Load Procedure

This procedure describes how to program a load target application procedure for the form. You might or might not want to allow users to download target applications to the target PC. However, if you do want to allow this action, you need to provide a control on the GUI for the user to do so. "Creating Event Procedures to Load Applications" on page 3-36 describes how to provide such a control.

1 In the project window, double-click the `Form` object.

   The code editor window opens.

2 In the procedure list box, select **Load**.

3 Create and initialize the objects for the `Load` method in the form.

```
Private Sub Form_Load()
    Set protocol_obj = New xPCProtocol
    Set target_obj = New xPCTarget
    Set scope_obj = New xPCScopes
    Set parameters_obj = New xpc_tank1pt
    Set signals_obj = New xpc_tank1bio

    stat = protocol_obj.Init
    stat = protocol_obj.RS232Connect(0, 0)
    stat = target_obj.Init(protocol_obj)
    stat = scope_obj.Init(protocol_obj)
    stat = parameters_obj.Init(protocol_obj.Ref)
    stat = signals_obj.Init(protocol_obj.Ref)
End Sub
```

You can add more code to the `Load` method. This is the minimum code you should enter for this method.

Your code editor window should look similar to the following.

```
Project1 - Form1 (Code)
Form                                          Load

Dim protocol_obj As xPCProtocol
Dim target_obj As xPCTarget
Dim scope_obj As xPCScopes

Dim parameters_obj As xpc_tank1pt
Dim signals_obj As xpc_tank1bio

Private Sub Form_Load()
    Set protocol_obj = New xPCProtocol
    Set target_obj = New xPCTarget
    Set scope_obj = New xPCScopes
    Set parameters_obj = New xpc_tank1pt
    Set signals_obj = New xpc_tank1bio

    stat = protocol_obj.Init
    stat = protocol_obj.RS232Connect(0, 0)
    stat = target_obj.Init(protocol_obj)
    stat = scope_obj.Init(protocol_obj)
    stat = parameters_obj.Init(protocol_obj.Ref)
    stat = signals_obj.Init(protocol_obj.Ref)
End Sub
```

Creating Event Procedures

Code in a Visual Basic application is divided into smaller blocks called *procedures*. Event procedures, such as those you create here, contain code that mainly calls the Target API component methods. For example, when a user clicks a button, that action starts the xPC Target application.

This code is also responsible for the feedback action (such as enabling a timer control, disabling/enabling controls) when an event occurs. An event procedure for a control combines the control's name (specified in the Name property), an underscore (_), and the event name. For example, if you want a command button named **Command1** to invoke an event procedure when it is clicked, call the procedure `Command1_Click`. The following procedures illustrate how to create event procedures, using the `xpctank` (or `xpc_tank1`) model as an example.

Creating Event Procedures to Load Applications

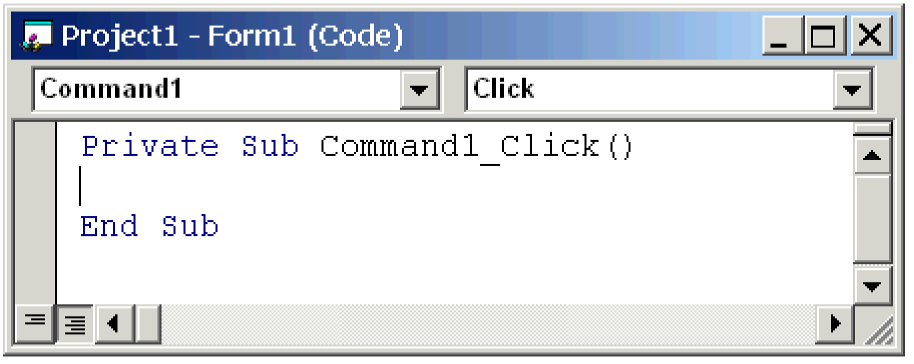
This procedure describes how to program the command button **Command1** to load an application to the target PC through a serial connection. Provide a procedure like this to allow users to download target applications to the target PC.

1. Double-click the form or, from the **View** menu, select **Code**.

2. From the object list box, select the name of an object in the active form. (The *active* form is the form that currently has the focus.) For this example, choose the command button **Command1**.

3. In the procedure list box, select the name of an event for the selected object.

   Here, the `Click` procedure is already selected because it is the default procedure for a command button.



4. To load the target application, enter the path to the target application. If the target application is in the same folder as the API application, enter ".". Enter the name of the target application without the extension.

   ```
   stat = target_obj.LoadApp(".", "xpc_tank1")
   ```

When you are done, the contents of your code editor window should look similar to the code below:

```
Private Sub Command1_Click()
    stat = target_obj.LoadApp(".", "xpc_tank1")
End Sub
```

Creating Event Procedures to Start and Stop Applications

This procedure describes how to program the command buttons **Command2** and **Command3** to start and stop an application on a target PC:

1. If you are not already in the code editor window, double-click the form or, from the **View** menu, select **Code**.

2. From the object list box, select the name of an object in the active form. (The *active* form is the form that currently has the focus.) For this example, choose the command button **Command2**.

3. In the procedure list box, select the name of an event for the selected object. Here, select the `Click` procedure.

4. To start the target application, select the `StartApp` method for the command button **Command2** (this is the button you named `Start`).

   ```
   stat = target_obj.StartApp
   ```

5. To stop the target application, select the `StopApp` method for the command button **Command3** (this is the button you named `Stop`). Be sure to select the `Click` procedure in the procedure list box.

   ```
   stat = target_obj.StopApp
   ```

When you are done, the contents of your code editor window should look similar to the code below:

```
Private Sub Command2_Click()
    stat = target_obj.StartApp
End Sub

Private Sub Command3_Click()
    stat = target_obj.StopApp
End Sub
```

Creating Event Procedures to Vary Input Values

You can provide controls to allow users to vary the parameters of their applications. The Scroll procedure is one way of varying input. The following code uses the Visual Basic `HScrollBar` object to vary the `water_level` parameter. It takes the value from the `HScrollBar` object and sends that value to the target as a parameter change.

Note This section assumes that you have tagged block parameters and created your own model-specific COM library. Refer to “Getting Parameter IDs with the GetParamIdx Method” on page 3-41 for a description of how to manually perform the equivalent of using tagged parameters.

---

1 If you are not already in the code editor window, double-click the form or, from the **View** menu, select **Code**.

2 From the object list box, select the name of an object in the active form. (The *active* form is the form that currently has the focus.) For this example, select the `HScroll1` object.

   The cursor jumps to the `HScroll1` object template of the code editor window.

3 In the procedure list box, select the name of an event for the selected object. Here, select the `Scroll` procedure.

4 Declare the `slideVal` variable as a double. The `slideVal` variable will contain the value of the scrollbar.

```
Dim slideVal(0) As Double
```

5 Assign to the `slideVal` variable the result of `CDbl`. The `CDbl` function reads the value of an object property. In this example, the object `HScroll1` has the property `slideVal(0)`. `CDbl` reads the value of `HScroll1.Value` and returns that value to `slideVal`.

```
slideVal(0) = CDbl(HScroll1.Value)
```

6 Set the value of `water_level` to the scroll bar value `slideVal`, which is from `HScrollBar`. The `COM` object `target_obj` has the method `SetParam`, which has the syntax `SetParam(parIdx, newparVal)`. The `SetParam` method references `parIdx` from the model-specific `COM` object (type `xpc_tank1pt`). To set the value of `water_level` to the scroll bar value `slideVal`, select `SetParam` and continue typing. A list of the parameters you tagged in

the Simulink model then pops up, and you can select the parameter water_level and continue typing.

The call to SetParam should look like the following:

```
stat = target_obj.SetParam(parameters_obj.water_level,
slideVal)
```

When you are done, the contents of your code editor window should look similar to the code below:

```
Private Sub HScroll1_Scroll()
    Dim slideVal(0) As Double

    slideVal(0) = CDbl(HScroll1.Value)
    stat = target_obj.SetParam(parameters_obj.water_level,
slideVal)
End Sub
```

Creating Event Procedures to Display Signal Values at the Host

You can provide controls to view signal values at the host. To do this, use a combination of the timer and label controls. The following code uses the Visual Basic timer control to display the water_level signal on the label control.

Note This section assumes that you have tagged signals and created your own model-specific COM library. Refer to "Getting Signal IDs with the GetSignalIdx Method" on page 3-43 for a description of how to manually perform the equivalent of using tagged signals.

Before you start, check that the Timer1 Interval property is greater than 0.

1 From the object list box, select the Timer1 object.

2 Assign to the Label1.Caption object the value of the water_level signal. The COM object target_obj has the method GetSignal(sigNum). Reference the sigNum parameter by passing it signals_obj.water_level. The CStr function converts the returned value to a string so that it can be displayed on the Label1 object.

When you are done, the contents of your code editor window should look similar to the code below:

```
Private Sub Timer1_Timer()
    Label1.Caption =
CStr(target_obj.GetSignal(signals_obj.water_level))
End Sub
```

Note Although you add both a timer and label object to the Visual Basic application, only the label appears on the GUI itself when the Visual Basic application is run. The timer is not visible.

Creating Unload and Termination Procedures

You should write Form Unload and Termination procedures to ensure that users are able to stop and unload the application appropriately, and to close the communication between the host PC and target PC.

Note Provide Form Unload and Termination procedures to ensure that the communication channel between the host PC and target PC properly closes between each run of the GUI application.

The Terminate procedure controls the behavior of the Visual Basic **Run** menu **End** option. The Unload procedure controls the behavior of the Visual Basic **Close** button.

1. From the object list box, select the `Form` object.
2. From the procedure list box, select `Terminate`.
3. You are going to close the connection with the target PC, so type `protocol_obj` and select the `Close` method for that object.

   ```
   protocol_obj.Close
   ```

4. From the procedure list box, select `Unload`.

5 Repeat step

When you are done, the contents of your code editor window should look similar to the code below:

```
Private Sub Form_Terminate()
   protocol_obj.Close
End Sub
Private Sub Form_Unload(Cancel As Integer)
   protocol_obj.Close
End Sub
```

Referencing Parameters and Signals Without Using Tags

The sample code in "Creating Event Procedures to Vary Input Values" on page 3-37 and "Creating Event Procedures to Display Signal Values at the Host" on page 3-39 illustrates how to reference parameters that you tagged before building the Simulink model. This section describes how to reference these same parameters and signals from the COM API application code if you did not opt to tag signals and parameters.

Getting Parameter IDs with the GetParamIdx Method

When working with parameters in the context of varying input values, you use the `SetParam` and `GetParamIdx` methods. The `SetParam` method has the syntax

```
SetParam(ByVal parIdx As Integer, ByRef newparVal As
System.Array) As Long
```

where **`parIdx`** is the identifier that corresponds to the parameter you want to set. To obtain the parameter ID, **`parIdx`**, for `SetParam`, you need to call the `GetParamIdx` method. This method has the syntax

```
GetParamIdx(ByVal blockName As String, ByVal paramName As
String) As Long
```

The following procedure describes how to obtain the appropriate `GetParamIdx` block name and parameter name for the Visual Basic `HScrollBar` object. You need to reference the block name and parameter from the `model_namept.m` file.

1 Open a DOS window.

2 Change the directory to the directory that contains your prebuilt model.

3 Open the file `model_namept.m`. For example, you can use the `notepad` text editor.

```
notepad xpc_tank1pt.m
```

The editor opens for that file. If you are not in the directory in which the `xpc_tank1pt.m` file resides, be sure to type the full path for `xpc_tank1pt.m`.

4 Search for and copy the string for the block of the parameter you want to reference. For the `xpc_tank1` example, search for the SetPoint block if you want to reference the water level. For example,

```
SetPoint
```

5 Return to the code editor window for your project.

6 In the line that contains the call to `GetParamIdx`, enter the path for the `blockName` variable.

7 Return to the editor window for `model_namept.m`.

8 Search for and copy the string for the name of the parameter you are interested in. For example,

```
Value
```

If you do not know the name of the block parameter you are interested in, refer to “Model and Block Parameters” of the Simulink Reference documentation.

9 Return to the code editor window for your project.

10 In the line that contains the call to `GetParamIdx`, enter the path for the `paramName` variable. For example,

```
stat = target_obj.SetParam(target_obj.GetParamIdx
("SetPoint", "Value"), slideVal)
```

When you are done, the contents of your code editor window should look similar to the code below:

```
Private Sub HScroll1_Scroll()
    Dim slideVal(0) As Double

    slideVal(0) = CDbl(HScroll1.Value)
    stat =
target_obj.SetParam(target_obj.GetParamIdx
("SetPoint", "Value"), slideVal)

End Sub
```

Note, if you want to retrieve the full block path and parameter name of a block, use the `GetParamName` method. The `GetParamName` method returns a variant data type object with two elements. The first element contains the full block path, the second element contains the parameter name. The following example illustrates how to use the `GetParamName` method to get the block path and parameter name:

```
Dim Pname As Variant
Pname=xpc_tank1.GetParamName(GetParamIdx(Idx)
BlockPathString=CStr(Pname(0))
ParameterNameString=CStr(Pname(1))
```

In this example,

- `Idx` is the index to a parameter.
- `BlockPathString` contains the full block path string.
- `ParameterNameString` contains the parameter name string.

Getting Signal IDs with the GetSignalIdx Method

When working with signals in the context of displaying signal values, you use the `GetSignal` and `GetSignalIdx` methods. The `GetSignal` method has the syntax

```
GetSignal(sigNum As Long) As Double
```

where `sigNum` is the identifier that corresponds to the signal you want to set.

To obtain the signal ID `sigNum` for `GetSignal`, you call the `GetSignalIdx` method. This method has the syntax

```
GetSignalIdx(sigName As String) As Long
```

The following procedure describes how to obtain the appropriate `GetSignalIdx` block name for the Visual Basic timer object. You need to reference the block name and signal from the `model_namebio.m` file.

1. Open a DOS window.
2. Change the directory to the directory that contains your prebuilt model.
3. Open the file `model_namebio.m`. For example,

   ```
   notepad xpc_tank1bio.m
   ```

   The editor opens for that file. If you are not in the directory in which the `xpc_tank1bio.m` file resides, be sure to type the full path for `xpc_tank1bio.m`.
4. Search for and copy the string for the block of the signal you want to reference. For the `xpc_tank1` example, search for the `TankLevel` block to reference the tank level. For example,

   ```
   TankLevel
   ```

5. Return to the code editor window for your project.
6. In the line that contains the call to `GetSignalIdx`, enter the path for the `SigName` variable.

When you are done, the contents of your code editor window should look similar to the code below:

```
Private Sub Timer1_Timer()
   Label1.Caption =
CStr(target_obj.GetSignal(target_obj.GetSignalIdx("TankLevel"
)))
End Sub
```

Testing the Visual Basic Application

While creating your Visual Basic application, you might want to see how the application is progressing. Visual Basic allows you to run your application while still in the Visual Basic project. From the Visual Basic task bar, you can click the **Run** button . Alternatively, you can follow the procedure:

1 If you have MATLAB and a target object connected, close the port. For example, at the MATLAB command line, type

```
tg.close
```

2 From within the project, go to the **Run** menu.

3 Select **Start** or **Start with Full Compile**. The **Start** option starts your application immediately. The **Start with Full Compile** option starts the application after compilation.

The form you are working on pops up. Test your application. Ensure that only one version of the application is running at any given time. To stop the application from within Visual Basic, you can click the **End** button from the task bar. Alternatively, you can go to the **Run** menu and select **End**.

Note If your Visual Basic application opens a communication channel between the host PC and the target PC for the target application, be sure to close that open channel between test runs of the Visual Basic application. Not doing so can cause subsequent runs of the Visual Basic application to fail. "Creating Unload and Termination Procedures" on page 3-40 describes how to write a procedure to disconnect from the target PC. If you want to return control to MATLAB, be sure to close the Visual Basic project first.

Building the Visual Basic Application

After you finish designing, programming, and testing your Visual Basic GUI application, build your application. You can later distribute the GUI application to users, who can then use it to work with target applications.

1 From within the project, go to the **File** menu.

2 Select **Make** `project_name_COM.exe`, where `project_name` is the name of the Visual Basic project you have been working on.

3 At the pop-up box, select the directory in which you want to save the executable. Optionally, you can also rename the executable.

   The compiler generates the `project_name_COM.exe` file in the specified directory.

Deploying the API Application

This section assumes that you have built your xPC Target application and your Visual Basic xPC Target COM GUI application. If you have not yet done so, refer to “Creating the Target Application and Model-Specific COM Library” on page 3-14 and “Building the Visual Basic Application” on page 3-45, respectively.

When distributing the Visual Basic model application to users, provide the following files:

- `project_name_COM.exe`, the executable for the Visual Basic application
- `model_name.dlm`

  Provide `model_name.dlm` if you expect the user to download the target application to the target PC. Ensure that you have enabled an application load event on the Visual Basic interface (refer to “Creating the Load Procedure” on page 3-33).

  If you expect that the target application is already loaded on the target PC when the user runs the Virtual Basic GUI application, you might not want him or her to be able to load the target application to the target PC.
- `model_nameCOMiface.dll`, if you tag the signals and parameters in the model
- `xpcapiCOM.dll`, the xPC Target COM API dynamic link library
- `xpcapi.dll`, the xPC Target API dynamic link library

Have the user ensure that all the files are located in the same directory before he or she executes the Visual Basic application.

You must also ensure that the user knows how to register the application-dependent dynamic link libraries (refer to "Registering Dependent Dynamic Link Libraries" on page 3-47).

To run the application and download an xPC Target application, users need to have `project_name_COM.exe` and `model_name.dlm`, if provided, in the same directory.

Registering Dependent Dynamic Link Libraries

This procedure uses `xpc_tank1` as an example.

1. Open a DOS window.

2. Change the directory to the directory containing the API application files.

3. From the directory in which `xpcapiCOM.dll` resides, register the xPC Target COM API DLL by typing

   ```
   regsvr32 xpcapiCOM.dll
   ```

   DOS displays the message

   ```
   DllRegisterServer in xpcapiCOM.dll succeeded
   ```

Creating a New Visual Basic Project Using Microsoft Visual 7.1 or 8.0

The procedures for the preceding topics apply to Microsoft Visual 6.0 ("Creating a New Visual Basic Project" on page 3-20). The procedures to use Microsoft Visual 7.1 (.NET 2003) and 8.0 are similar, with the following exceptions. Note that references to Microsoft Visual 7.1 or .NET 2003 also apply to Microsoft Visual 8.0.

- You can open a Microsoft Visual 6.0 project under Microsoft Visual .NET 2003. Microsoft Visual .NET 2003 automatically converts the project.
- If you first create a new Visual Basic project, select **Windows Application** as the template.
- When referencing the xPC Target COM API and model-specific COM libraries, do the following

a From the **Project** menu, click **Add Reference**.

   The Add Reference dialog box opens.

b Select the **COM** tab.

c Scroll down the **Component Name** list to the bottom and select the **xPC Target API COM Type Library** item.

d Click **Select**.

   xPC Target API COM Type Library appears in the **Selected Components** pane.

e Click **OK**.

- When creating a reference to the xPC Target interface objects, include the COM library. The following illustrates example code on how to reference these objects in Microsoft Visual .NET 2003 and Microsoft Visual 6.0:

  Microsoft Visual .NET 2003

```
Public protocol_obj As XPCAPICOMLib.xPCProtocol
Public target_obj As XPCAPICOMLib.xPCTarget
Public scope_obj As XPCAPICOMLib.xPCScopes
```

  Microsoft Visual 6.0

```
Public protocol_obj As xPCProtocol
Public target_obj As xPCTarget
Public scope_obj As xPCScopes
```

- When creating an instance of the xPC Target interface objects, include the COM library. The following illustrates example code on how to create an instance of these objects in Microsoft Visual .NET 2003 and Microsoft Visual 6.0:

  Microsoft Visual .NET 2003

```
protocol_obj = New XPCAPICOMLib.xPCProtocol
target_obj = New XPCAPICOMLib.xPCTarget
scope_obj = New XPCAPICOMLib.xPCScopes
```

Microsoft Visual 6.0:

```
Set protocol_obj = New xPCProtocol
Set target_obj = New xPCTarget
Set scope_obj = New xPCScopes
```

- Microsoft Visual .NET 2003 builds applications into the **bin** directory of your project area. You cannot choose another location to place your executable.
- When distributing the Visual Basic model application to users, provide the following files in addition to those listed in "Deploying the API Application" on page 3-46:
  - `Interop.model_nameACOMIFACELib.dll`
  - `Interop.XPCAPICOMLib.dll`

xPC Target COM API Demos and Scripts

To help you better understand and quickly begin to use COM API functions to create custom GUI applications, xPC Target provides a number of API demos and scripts in the `C:\matlabroot\toolbox\rtw\targets\xpc\api` directory. This topic briefly describes those demos and scripts.

Microsoft Visual Basic 7.1 (.NET 2003) Demo (p. 4-2)	The Microsoft Visual Basic .NET 2003 demo illustrates how to create a generic custom GUI that connects to a target PC with any downloaded target application.
Microsoft Visual Basic 6.0 Demo (p. 4-5)	The Microsoft Visual Basic 6.0 sf_car_xpc demo illustrates how to create a custom GUI that connects to a target PC that has a specific (sf_car_xpc) downloaded target application.
Tcl/Tk Scripts (p. 4-8)	The Tcl/Tk demos are scripts that illustrate how to directly access COM API functions through a command-line interpreter like Tcl/Tk.

Microsoft Visual Basic 7.1 (.NET 2003) Demo

The Microsoft Visual Basic .NET 2003 demo illustrates how to create a custom GUI that connects to a target PC with a downloaded target application. The solution file for this demo is located in

C:*matlabroot*\toolbox\rtw\targets\xpc\api\VBNET\SigsAndParamsDemo

- `bin` — Contains the executable for the demo project and the `xpcapi.dll` file
- `Demo.sln` — Contains a solution file for the `Demo` project

The `Demo.sln` file contains all the Visual Basic .NET 2003 files to run the windows form application. This demo is a functional application that you can use as a template to create your own custom GUIs.

The COM API example from "Example Visual Basic GUI Using COM Objects" on page 3-3 is a simple GUI that illustrates some basic concepts for creating a GUI with the COM API. The `Demo` solution is a more advanced example that illustrates how to create a GUI similar to the xPC Target Explorer. The `Demo` solution is fully commented.

This demo illustrates how you can use the COM API to create a GUI that

- Connects to the target PC via an RS-232 or TCP/IP connection
- Starts and stops the target application loaded on the target PC
- Retrieves and lists all the signals in the target application
- Displays the value of a selected signal
- Retrieves and lists all the parameters in the target application
- Change the values of the parameters

This section describes the following:

- "Before Starting" on page 4-3
- "Accessing the Demo Project Solution" on page 4-3
- "Rebuilding the Demo Project Solution" on page 4-4

- "Using the Demo Executable" on page 4-4

Before Starting

To use the `Demo` solution, you need

- A target PC running a current xPC Target kernel
- A host PC running MATLAB, connected to the target PC via RS-232 or TCP/IP
- A target application loaded on the target PC

xPC Target ships with an executable version of the demo. If you want to rebuild the `Demo` solution, of if you want to write your own custom GUIs like this one, you need Microsoft Visual Basic .NET 2003 installed on the host PC.

Note xPC Target allows you to create applications, such as GUIs, to interact with a target PC with COM API functions. Chapter 3, "xPC Target COM API" describes this in detail. To deploy a GUI application to other host PC systems that do not have your licensed copy of xPC Target, you need the xPC Target Embedded Option. If you do not have the xPC Target Embedded Option and would like to deploy your GUI application, contact your MathWorks representative.

Accessing the Demo Project Solution

To access the `Demo` solution,

1. Copy the contents of the `VBNET` directory to a writable directory of your choice.

2. Change directory to the one that contains your copy of the `Demo` solution.

3. Double-click `demo.sln`.

   The Microsoft Development Environment for Visual Basic application starts.

4. In the **Solution Explorer** pane, double-click `Form1.vb` to display the `Demo` solution form.

The form is displayed. You can inspect the layout of the demo.

5 To inspect the form code, select the **View** menu `Code` option.

The Visual Basic code for the form is displayed.

Rebuilding the Demo Project Solution

To rebuild the `Demo` solution,

1 Double-click `demo.sln`.

The Microsoft Development Environment for Visual Basic application starts.

2 Select the **Build** menu `Build Solution` option.

Using the Demo Executable

To use the `Demo` solution executable,

1 Change directory to the one that contains your copy of the `Demo` solution.

2 Change directory to the `bin` directory.

3 Double-click `Demo1.exe`.

The GUI is displayed.

Microsoft Visual Basic 6.0 Demo

The Microsoft Visual Basic 6.0 `sf_car_xpc` demo illustrates how to create a custom GUI that connects to a target PC. The files for this demo are located in

```
C:\matlabroot\toolbox\rtw\targets\xpc\api\VisualBasicModels\sf_car_xpc
```

This application interfaces with the xPC Target application `sf_car_xpc.dlm`, built from the Simulink model `sf_car_xpc.mdl`. This model simulates an automatic transmission control system composed of modules that represent the engine, transmission, and vehicle, with an additional logic block to control the transmission ratio. User inputs to the model are in the form of throttle (%) and brake torque (ft-lb).

This demo illustrates how you can use the COM API to create a GUI that

- Connects to the target PC via an RS-232 or TCP/IP connection
- Loads the `sf_car_xpc.dlm` target application to the target PC
- Starts and starts the target application engine
- Edits the stop time of the target application
- Edits the sample time of the target application
- Displays the speed, RPM, and gear of the target application engine

---

Note For detailed information on the project, see the `readme.txt` file located in `C:\`*`matlabroot`*`\toolbox\rtw\targets\xpc\api\VisualBasic\Models\sf_car_xpc`.

This section describes the following:

- "Before Starting" on page 4-6
- "Accessing the sf_car_xpc Project" on page 4-6
- "Rebuilding the sf_car_xpc Project" on page 4-7
- "Using the sf_car_xpc Executable" on page 4-7

Before Starting

To use the sf_car_xpc project, you need

- A target PC running a current xPC Target kernel
- A host PC running MATLAB, connected to the target PC via RS-232 or TCP/IP

xPC Target ships with an executable version of the sf_car_xpc project. If you want to rebuild the sf_car_xpc project, you need Microsoft Visual Basic 6.0 Professional installed on the host PC. If you want to view or edit the model, you need to have the Stateflow® product installed on the host PC.

Note xPC Target allows you to create applications, such as GUIs, to interact with a target PC with COM API functions. Chapter 3, "xPC Target COM API" describes this in detail. To deploy a GUI application to other host PC systems that do not have your licensed copy of xPC Target, you need the xPC Target Embedded Option. If you do not have the xPC Target Embedded Option and would like to deploy your GUI application, contact your MathWorks representative.

Accessing the sf_car_xpc Project

To access the sf_car_xpc project,

1. Copy the contents of the VisualBasic directory to a writable directory of your choice.
2. Change directory to the one that contains your copy of the sf_car_xpc project.
3. Double-click the Visual Basic project. For example, double-click sf_car_xpc_COM.vbp.

   The Microsoft Visual Basic application starts.
4. In the right **Project** pane, expand the Forms folder.
5. Double-click the form you want to look at.

The form is displayed. You can inspect the layout of it.

6 To inspect the form code, select the **View** menu `Code` option.

The Visual Basic code for the form is displayed.

Rebuilding the sf_car_xpc Project

To rebuild the `sf_car_xpc` project,

1 Double-click the Visual Basic project. For example, double-click `sf_car_xpc_COM.vbp`.

The Microsoft Visual Basic application starts.

2 Select the **File** menu `Make sf_car_xpc.exe`.

Using the sf_car_xpc Executable

To use the `sf_car_xpc` project executable,

1 Change directory to the one that contains your copy of the `sf_car_xpc` project.

2 Change directory to the `bin` directory.

3 Double-click `sf_car_xpc.exe`.

The GUI is displayed.

Tcl/Tk Scripts

The Tcl/Tk demos are scripts that illustrate how to directly access xPC Target COM API functions through a command-line interpreter like Tcl/Tk. With Tcl/Tk

- You can write simple command-line scripts that communicate with a target PC and the target application downloaded on that target PC.
- You can write simple GUIs that you can use to interact with a target application downloaded on a target PC.

The files for this scripts are located in

`C:\`*`matlabroot`*`\toolbox\rtw\targets\xpc\api\tcltk`

- `xpcapi.dll` — The xPC Target API DLL file. This file must be in the current (`pwd`) directory. Alternatively, you can copy the file `xpcapi.dll` into the Windows system directory.
- `xpcbase.tcl` — Contains utility procedures used by the other scripts in the series
- `xpclists.tcl` — Generates a list of signals or parameters for the target application currently loaded on the target PC
- `xpcload.tcl` — Loads the specified target application to the connected target PC
- `xpcoutputlog.tcl` — Reads log data from the target PC and plots the data on the host PC
- `xpcstart.tcl` — Starts the target application loaded on the target PC
- `xpcstop.tcl` — Stops the target application loaded on the target PC
- `xpctargetping.tcl` — Tests the communication between the host and target PCs
- `xpctargetscope.tcl` — Creates a simple GUI that enables you to add and control a scope of type target
- `xpctune.tcl` — Creates a simple GUI slider that enables you to manipulate a parameter value for the target PC application

This section describes the following:

- "Required Tcl/Tk Software" on page 4-9
- "Using the Demo Scripts" on page 4-9

Required Tcl/Tk Software

To use these Tcl/Tk scripts, or to write your own Tcl/Tk scripts, you need

- An installation of a Tcl/Tk distribution on the host PC.
- An add-on package to the Tcl/Tk interpreter so that the scripts can access the COM API objects. The `tcom` package is recommended. This package was used to create the demo scripts in the `C:\`*`matlabroot`*`\toolbox\rtw\targets\tcltk` directory.
- The `math::statistics` package. This package is required for the `xpcoutputlog.tcl` file.

Note There are Tcl/Tk distributions that include required and useful packages for use with xPC Target. For example, the Tcl/Tk distribution at `http://www.activestate.com` contains these packages.

Using the Demo Scripts

The top of each Tcl/Tk script file contains directions on how to use each Tcl/Tk scripts. In general:

1. Copy the contents of the `tcltk` directory to a writable directory of your choice.
2. Change directory to the one that contains your copy of the Tcl/Tk script files.
3. Start your Tcl/Tk interpreter.
4. Load the Tcl/Tk script with the `source` command. For example,

   ```
   source xpctargetping.tcl
   ```

5. Run the loaded script. For example,

```
xpctargetping 192.168.0.10 22222
```

The selected script executes. In this example, `xpctargetping.tcl` tests the communication between the host and target PC and returns a success or failure message.

API Functions and Methods — By Category

C API Functions

Logging, Scope, and File System Structures

`dirStruct`	Type definition for file system directory information structure
`diskinfo`	Type definition for file system disk information structure
`lgmode`	Type definition for logging options structure
`scopedata`	Type definition for scope data structure

Communications Functions

xPCCloseConnection	Close RS-232 or TCP/IP communication connection
xPCClosePort	Close RS-232 or TCP/IP communication connection
xPCDeRegisterTarget	Delete target communication properties from xPC Target API library
xPCGetLoadTimeOut	Return initialization time-out value of target application
xPCOpenConnection	Open connection to target PC
xPCOpenSerialPort	Open RS-232 connection to xPC Target system
xPCOpenTcpIpPort	Open TCP/IP connection to xPC Target system
xPCReboot	Reboot target PC
xPCRegisterTarget	Register target with xPC Target API library
xPCReOpenPort	Reopen communication channel
xPCSetLoadTimeOut	Change initialization time-out value
xPCTargetPing	Ping target PC

Target Application Functions

xPCAverageTET	Return average task execution time
xPCGetAPIVersion	Get version number of xPC Target API
xPCGetAppName	Return target application name
xPCGetExecTime	Return target application execution time

`xPCGetSampleTime`	Return target application sample time
`xPCGetStopTime`	Return stop time
`xPCGetTargetVersion`	Get xPC Target kernel version
`xPCIsAppRunning`	Return target application running status
`xPCIsOverloaded`	Return target PC overload status
`xPCLoadApp`	Load target application onto target PC
`xPCLoadParamSet`	Restore parameter values
`xPCMaximumTET`	Copy maximum task execution time to array
`xPCMinimumTET`	Copy minimum task execution time to array
`xPCSaveParamSet`	Save parameter values of target application
`xPCSetSampleTime`	Change target application sample time
`xPCSetStopTime`	Change target application stop time
`xPCStartApp`	Start target application
`xPCStopApp`	Stop target application
`xPCUnloadApp`	Unload target application

Data Logging Functions

`xPCGetLogMode`	Return logging mode and increment value for target application
`xPCGetNumOutputs`	Return number of outputs
`xPCGetNumStates`	Return number of states
`xPCGetOutputLog`	Copy output log data to array

`xPCGetStateLog`	Copy state log values to array
`xPCGetTETLog`	Copy TET log to array
`xPCGetTimeLog`	Copy time log to array
`xPCMaxLogSamples`	Return maximum number of samples that can be in log buffer
`xPCNumLogSamples`	Return number of samples in log buffer
`xPCNumLogWraps`	Return number of times log buffer wraps
`xPCSetLogMode`	Set logging mode and increment value of scope

Scope Functions

`xPCAddScope`	Create new scope
`xPCGetScope`	Get and copy scope data to structure
`xPCGetScopes`	Get and copy list of scope numbers
`xPCIsScFinished`	Return data acquisition status for scope
`xPCRemScope`	Remove scope
`xPCScAddSignal`	Add signal to scope
`xPCScGetData`	Copy scope data to array
`xPCScGetDecimation`	Return decimation of scope
`xPCScGetNumPrePostSamples`	Get number of pre- or posttriggering samples before triggering scope
`xPCScGetNumSamples`	Get number of samples in one data acquisition cycle
`xPCScGetSignals`	Copy list of signals to array
`xPCScGetStartTime`	Get start time for last data acquisition cycle

`xPCScGetState`	Get state of scope
`xPCScGetTriggerLevel`	Get trigger level for scope
`xPCScGetTriggerMode`	Get trigger mode for scope
`xPCScGetTriggerScope`	Get trigger scope
`xPCScGetTriggerScopeSample`	Get sample number for triggering scope
`xPCScGetTriggerSignal`	Get trigger signal for scope
`xPCScGetTriggerSlope`	Get trigger slope for scope
`xPCScGetType`	Get type of scope
`xPCScRemSignal`	Remove signal from scope
`xPCScSetDecimation`	Set decimation of scope
`xPCScSetNumPrePostSamples`	Set number of pre- or posttriggering samples before triggering scope
`xPCScSetNumSamples`	Set number of samples in one data acquisition cycle
`xPCScSetTriggerLevel`	Set trigger level for scope
`xPCScSetTriggerMode`	Set trigger mode of scope
`xPCScSetTriggerScope`	Select scope to trigger another scope
`xPCScSetTriggerScopeSample`	Set sample number for triggering scope
`xPCScSetTriggerSignal`	Select signal to trigger scope
`xPCScSetTriggerSlope`	Set slope of signal that triggers scope
`xPCScSoftwareTrigger`	Set software trigger of scope
`xPCScStart`	Start data acquisition for scope
`xPCScStop`	Stop data acquisition for scope
`xPCSetScope`	Set properties of scope

File System Functions

`xPCFSCD`	Change current directory on target PC to specified path
`xPCFSCloseFile`	Close file on target PC
`xPCFSDir`	Get contents of specified directory on target PC
`xPCFSDirItems`	Get contents of specified directory on target PC
`xPCFSDirSize`	Return size of specified directory on target PC
`xPCFSDirStructSize`	Get number of items in directory
`xPCFSDiskInfo`	Information about target PC file system
`xPCFSGetError`	Get text description for error number on target PC file system
`xPCFSGetFileSize`	Return size of file on target PC
`xPCFSGetPWD`	Get current directory of target PC
`xPCFSOpenFile`	Open file on target PC
`xPCFSReadFile`	Read open file on target PC
`xPCFSRemoveFile`	Remove file from target PC
`xPCFSRMDIR`	Remove directory from target PC
`xPCFSScGetFilename`	Get name of file for scope
`xPCFSScGetWriteMode`	Get write mode of file for scope
`xPCFSScGetWriteSize`	Get block write size of data chunks
`xPCFSScSetFilename`	Specify name for file to contain signal data
`xPCFSScSetWriteMode`	Specify when file allocation table entry is updated

xPCFSScSetWriteSize	Specify that memory buffer collect data in multiples of write size
xPCFSWriteFile	Write to file on target PC

Target Scope Functions

xPCTgScGetGrid	Get status of grid line for particular scope
xPCTgScGetMode	Get scope mode for displaying signals
xPCTgScGetViewMode	Get view mode for target PC display
xPCTgScGetYLimits	Copy *y*-axis limits for scope to array
xPCTgScSetGrid	Set grid mode for scope
xPCTgScSetMode	Set display mode for scope
xPCTgScSetViewMode	Set view mode for scope
xPCTgScSetYLimits	Set *y*-axis limits for scope

Monitoring and Tuning Functions

xPCGetNumParams	Return number of tunable parameters
xPCGetNumSignals	Return number of signals
xPCGetParam	Get parameter value and copy it to array
xPCGetParamDims	Get row and column dimensions of parameter
xPCGetParamIdx	Return parameter index
xPCGetParamName	Get name of parameter
xPCGetSigIdxfromLabel	Return array of signal indices
xPCGetSigLabelWidth	Return number of elements in signal
xPCGetSignal	Return value of signal

`xPCGetSignalIdx`	Return index for signal
`xPCGetSignalName`	Copy name of signal to character array
`xPCGetSignals`	Return vector of signal values
`xPCGetSignalWidth`	Return width of signal
`xPCSetParam`	Change value of parameter

Miscellaneous Functions

`xPCErrorMsg`	Return text description for error message
`xPCFreeAPI`	Unload xPC Target DLL
`xPCGetEcho`	Return display mode for target message window
`xPCGetLastError`	Return constant of last error
`xPCInitAPI`	Initialize xPC Target DLL
`xPCSetEcho`	Turn message display on or off
`xPCSetLastError`	Set last error to specific string constant

COM API Methods

Communication Objects (xPCProtocol)

`xPCProtocol.Close`	Close RS-232 or TCP/IP communication connection
`xPCProtocol.GetLoadTimeOut`	Return current time-out value for target application initialization
`xPCProtocol.GetxPCErrorMsg`	Return error string
`xPCProtocol.Init`	Initialize xPC Target API DLL
`xPCProtocol.isxPCError`	Return error status
`xPCProtocol.Port`	Contain communication channel index
`xPCProtocol.Reboot`	Reboot target PC
`xPCProtocol.RS232Connect`	Open RS-232 connection to target PC
`xPCProtocol.SetLoadTimeOut`	Change initialization time-out value
`xPCProtocol.TargetPing`	Ping target PC
`xPCProtocol.TcpIpConnect`	Open TCP/IP connection to target PC
`xPCProtocol.Term`	Unload xPC Target API DLL from memory

Scope Objects (xPCScopes)

`xPCScopes.AddFileScope`	Create new scope of type file
`xPCScopes.AddHostScope`	Create new scope of type host
`xPCScopes.AddTargetScope`	Create new scope of type target
`xPCScopes.GetScopes`	Get and copy list of scope numbers
`xPCScopes.GetxPCError`	Get error string
`xPCScopes.Init`	Initialize scope object to communicate with target PC
`xPCScopes.IsScopeFinished`	Get data acquisition status for scope
`xPCScopes.isxPCError`	Get error status
`xPCScopes.RemScope`	Remove scope
`xPCScopes.ScopeAddSignal`	Add signal to scope
`xPCScopes.ScopeGetData`	Copy scope data to array
`xPCScopes.ScopeGetDecimation`	Get decimation of scope
`xPCScopes.ScopeGetNumPrePost-Samples`	Get number of pre- or posttriggering samples before triggering scope
`xPCScopes.ScopeGetNumSamples`	Get number of samples in one data acquisition cycle
`xPCScopes.ScopeGetSignals`	Get list of signals
`xPCScopes.ScopeGetStartTime`	Get last data acquisition cycle start time
`xPCScopes.ScopeGetState`	Get state of scope
`xPCScopes.ScopeGetTriggerLevel`	Get trigger level for scope
`xPCScopes.ScopeGetTriggerMode`	Get trigger mode for scope
`xPCScopes.ScopeGetTriggerModeStr`	Get trigger mode as string
`xPCScopes.ScopeGetTriggerSample`	Get sample number for triggering scope
`xPCScopes.ScopeGetTriggerSignal`	Get trigger signal for scope

`xPCScopes.ScopeGetTriggerSlope`	Get trigger slope for scope
`xPCScopes.ScopeGetTriggerSlope-Str`	Get trigger slope as string
`xPCScopes.ScopeGetType`	Get type of scope
`xPCScopes.ScopeRemSignal`	Remove signal from scope
`xPCScopes.ScopeSetDecimation`	Set decimation of scope
`xPCScopes.ScopeSetNumPrePost-Samples`	Set number of pre- or posttriggering samples before triggering scope
`xPCScopes.ScopeSetNumSamples`	Set number of samples in one data acquisition cycle
`xPCScopes.ScopeSetTriggerLevel`	Set trigger level for scope
`xPCScopes.ScopeSetTriggerMode`	Set trigger mode of scope
`xPCScopes.ScopeSetTriggerSample`	Set sample number for triggering scope
`xPCScopes.ScopeSetTriggerSignal`	Select signal to trigger scope
`xPCScopes.ScopeSetTriggerSlope`	Set slope of signal that triggers scope
`xPCScopes.ScopeSoftwareTrigger`	Set software trigger of scope
`xPCScopes.ScopeStart`	Start data acquisition for scope
`xPCScopes.ScopeStop`	Stop data acquisition for scope
`xPCScopes.TargetScopeGetGrid`	Get status of grid line for particular scope
`xPCScopes.TargetScopeGetMode`	Get scope mode for displaying signals
`xPCScopes.TargetScopeGetModeStr`	Get scope mode string for displaying signals
`xPCScopes.TargetScopeGetViewMode`	Get view mode for target PC display
`xPCScopes.TargetScopeGetYLimits`	Get *y*-axis limits for scope
`xPCScopes.TargetScopeSetGrid`	Set grid mode for scope
`xPCScopes.TargetScopeSetMode`	Set display mode for scope

`xPCScopes.TargetScopeSetViewMode`	Set view mode for scope
`xPCScopes.TargetScopeSetYLimits`	Set y-axis limits for scope

Target Objects (xPCTarget)

`xPCTarget.AverageTET`	Get average task execution time
`xPCTarget.GetAppName`	Get target application name
`xPCTarget.GetExecTime`	Get execution time for target application
`xPCTarget.GetNumOutputs`	Get number of outputs
`xPCTarget.GetNumParams`	Get number of tunable parameters
`xPCTarget.GetNumSignals`	Get number of signals
`xPCTarget.GetNumStates`	Get number of states
`xPCTarget.GetOutputLog`	Copy output log data to array
`xPCTarget.GetParam`	Get parameter values
`xPCTarget.GetParamDims`	Get row and column dimensions of parameter
`xPCTarget.GetParamIdx`	Get parameter index
`xPCTarget.GetParamName`	Get parameter name
`xPCTarget.GetSampleTime`	Get sample time
`xPCTarget.GetSignal`	Get signal value
`xPCTarget.GetSignalidsfromLabel`	Get signal IDs from signal label
`xPCTarget.GetSignalIdx`	Get signal index
`xPCTarget.GetSignalLabel`	Get signal label
`xPCTarget.GetSignalName`	Copy signal name to character array
`xPCTarget.GetSignalWidth`	Get width of signal
`xPCTarget.GetStateLog`	Get state log
`xPCTarget.GetStopTime`	Get stop time

`xPCTarget.GetTETLog`	Get TET log
`xPCTarget.GetTimeLog`	Get time log
`xPCTarget.GetxPCError`	Get error string
`xPCTarget.Init`	Initialize target object to communicate with target PC
`xPCTarget.IsAppRunning`	Return running status for target application
`xPCTarget.IsOverloaded`	Return overload status for target PC
`xPCTarget.isxPCError`	Return error status
`xPCTarget.LoadApp`	Load target application onto target PC
`xPCTarget.MaximumTET`	Copy maximum task execution time to array
`xPCTarget.MaxLogSamples`	Return maximum number of samples that can be in log buffer
`xPCTarget.MinimumTET`	Copy minimum task execution time to array
`xPCTarget.NumLogSamples`	Return number of samples in log buffer
`xPCTarget.NumLogWraps`	Return number of times log buffer wraps
`xPCTarget.SetParam`	Change parameter value
`xPCTarget.SetSampleTime`	Change sample time for target application
`xPCTarget.SetStopTime`	Change stop time of target application
`xPCTarget.StartApp`	Start target application
`xPCTarget.StopApp`	Stop target application
`xPCTarget.UnLoadApp`	Unload target application

File System Objects (xPCFileSystem)

`FSDir`	Type definition for file system directory information structure
`FSDiskInfo`	Type definition for file system disk information structure
`xPCFileSystem.CD`	Change current directory on target PC to specified path
`xPCFileSystem.CloseFile`	Close file on target PC
`xPCFileSystem.DirList`	Return contents of target PC directory
`xPCFileSystem.GetDiskInfo`	Return disk information
`xPCFileSystem.GetFileSize`	Return size of file on target PC
`xPCFileSystem.Init`	Initialize file system object to communicate with target PC
`xPCFileSystem.MKDIR`	Create directory on target PC
`xPCFileSystem.OpenFile`	Open file on target PC
`xPCFileSystem.PWD`	Get current directory of target PC
`xPCFileSystem.ReadFile`	Read open file on target PC
`xPCFileSystem.RemoveFile`	Remove file from target PC
`xPCFileSystem.RMDIR`	Remove directory from target PC
`xPCFileSystem.ScGetFileName`	Get name of file for scope
`xPCFileSystem.ScGetWriteMode`	Get write mode of file for scope
`xPCFileSystem.ScGetWriteSize`	Get block write size of data chunks
`xPCFileSystem.ScSetFileName`	Specify file name to contain signal data
`xPCFileSystem.ScSetWriteMode`	Specify when file allocation table entry is updated

`xPCFileSystem.ScSetWriteSize`	Specify that memory buffer collect data in multiples of write size
`xPCFileSystem.WriteFile`	Write to file on target PC

6

API Functions and Methods — Alphabetical List

dirStruct

Purpose

Type definition for file system directory information structure

Prototype

```
typedef struct {
   char          Name[8];
   char          Ext[3];
   char          Day;
   int  Month;
   int  Year;
   int  Hour;
   int  Min;
   int  isDir;
   unsigned long  Size;
} dirStruct;
```

Arguments

Name	This value contains the name of the file or directory.
Ext	This value contains the file type of the element, if the element is a file (*isDir* is 0). If the element is a directory (*isDir* is 1), this field is empty.
Day	This value contains the day the file or directory was last modified.
Month	This value contains the month the file or directory was last modified.
Year	This value contains the year the file or directory was last modified.
Hour	This value contains the hour the file or directory was last modified.
Min	This value contains the minute the file or directory was last modified.

isDir	This value indicates if the element is a file (0) or directory (1). If it is a directory, Bytes has a value of 0.
Size	This value contains the size of the file in bytes. If the element is a directory, this value is 0.

Description

The dirStruct structure contains information for a directory in the file system.

See Also

API function xPCFSDirItems

diskinfo

Purpose

Type definition for file system disk information structure

Prototype

```
typedef struct {
   char         Label[12];
   char         DriveLetter;
   char         Reserved[3];
   unsigned int SerialNumber;
   unsigned int FirstPhysicalSector;
   unsigned int FATType;
   unsigned int FATCount;
   unsigned int MaxDirEntries;
   unsigned int BytesPerSector;
   unsigned int SectorsPerCluster;
   unsigned int TotalClusters;
   unsigned int BadClusters;
   unsigned int FreeClusters;
   unsigned int Files;
   unsigned int FileChains;
   unsigned int FreeChains;
   unsigned int LargestFreeChain;
} diskinfo;
```

Arguments

Label	This value contains the zero-terminated string that contains the volume label. The string is empty if the volume has no label.
DriveLetter	This value contains the drive letter, in uppercase.
Reserved	Reserved.
SerialNumber	This value contains the volume serial number.
FirstPhysicalSector	This value contains the logical block addressing (LBA) address of the logical drive boot record. For 3.5-inch disks, this value is `0`.

`FATType`	This value contains the type of file system found. It can contain 12 , 16 , or 32 for FAT-12, FAT-16, or FAT-32 volumes, respectively.
`FATCount`	This value contains the number of FAT partitions on the volume.
`MaxDirEntries`	This value contains the size of the root directory. For FAT-32 systems, this value is 0.
`BytesPerSector`	This value contains the sector size. This value is most likely to be `512`.
`SectorsPerCluster`	This value contains, in sectors, the size of the smallest unit of storage that can be allocated to a file.
`TotalClusters`	This value contains the number of file storage clusters on the volume.
`BadClusters`	This value contains the number of clusters that have been marked as bad. These clusters are unavailable for file storage.
`FreeClusters`	This value contains the number of clusters that are currently available for storage.
`Files`	This value contains the number of files, including directories, on the volume. This number excludes the root directory and files that have an allocated file size of 0.
`FileChains`	This value contains the number of contiguous cluster chains. On a completely unfragmented volume, this value is identical to the value of `Files`.

FreeChains	This value contains the number of contiguous cluster chains of free clusters. On a completely unfragmented volume, this value is 1.
LargestFreeChain	This value contains the maximum allocated file size, in number of clusters, for a newly allocated contiguous file. On a completely unfragmented volume, this value is identical to `FreeClusters`.

Description

The `diskinfo` structure contains information for file system disks.

See Also

API function `xPCFSDiskInfo`

Purpose

Type definition for file system directory information structure

Prototype

```
typedef struct {
BSTR Name;
BSTR Date;
BSTR Time;
long Bytes;
long isdir;
} FSDir;
```

Arguments

Name	This value contains the name of the file or directory.
Date	This value contains the date the file or directory was last modified.
Time	This value contains the time the file or directory was last modified.
Bytes	This value contains the size of the file in bytes. If the element is a directory, this value is 0.
isdir	This value indicates if the element is a file (0) or directory (1). If it is a directory, *Bytes* has a value of 0.

Description

The FSDir structure contains information for a directory in the file system.

See Also

API methodxPCFileSystem.DirList

FSDiskInfo

Purpose

Type definition for file system disk information structure

Prototpye

```
typedef struct {
   BSTR Label;
   BSTR DriveLetter;
   BSTR Reserved;
   long SerialNumber;
   long FirstPhysicalSector;
   long FATType;
   long FATCount;
   long MaxDirEntries;
   long BytesPerSector;
   long SectorsPerCluster;
   long TotalClusters;
   long BadClusters;
   long FreeClusters;
   long Files;
   long FileChains;
   long FreeChains;
   long LargestFreeChain;
} FSDiskInfo;
```

Arguments

Label	This value contains the zero-terminated string that contains the volume label. The string is empty if the volume has no label.
DriveLetter	This value contains the drive letter, in uppercase.
Reserved	Reserved.
SerialNumber	This value contains the volume serial number.
FirstPhysicalSector	This value contains the logical block address (LBA) of the logical drive boot record. For 3.5-inch disks, this value is `0`.

FATType	This value contains the type of file system found. It can contain 12 , 16 , or 32 for FAT-12, FAT-16, or FAT-32 volumes, respectively.
FATCount	This value contains the number of FAT partitions on the volume.
MaxDirEntries	This value contains the size of the root directory. For FAT-32 systems, this value is 0.
BytesPerSector	This value contains the sector size. This value is most likely to be `512`.
SectorsPerCluster	This value contains, in sectors, the size of the smallest unit of storage that can be allocated to a file.
TotalClusters	This value contains the number of file storage clusters on the volume.
BadClusters	This value contains the number of clusters that have been marked as bad. These clusters are unavailable for file storage.
FreeClusters	This value contains the number of clusters that are currently available for storage.
Files	This value contains the number of files, including directories, on the volume. This number excludes the root directory and files that have an allocated file size of 0.
FileChains	This value contains the number of contiguous cluster chains. On a completely unfragmented volume, this value is identical to the value of *Files*.

`FreeChains`	This value contains the number of contiguous cluster chains of free clusters. On a completely unfragmented volume, this value is 1.
`LargestFreeChain`	This value contains the maximum allocated file size, in number of clusters, for a newly allocated contiguous file. On a completely unfragmented volume, this value is identical to *`FreeClusters`*.

Description

The `FSDiskInfo` structure contains information for file system disks.

See Also

API method `xPCFileSystem.GetDiskInfo`

Purpose

Type definition for logging options structure

Prototype

```
typedef struct {
   int    mode;
   double incrementvalue;
} lgmode;
```

Arguments

mode	This value indicates the type of logging you want. Specify LGMOD_TIME for time-equidistant logging. Specify LGMOD_VALUE for value-equidistant logging.
incrementvalue	If you set *mode* to LGMOD_VALUE for value-equidistant data, this option specifies the increment (difference in amplitude) value between logged data points. A data point is logged only when an output signal or a state changes by *incrementvalue*. If you set *mode* to LGMOD_TIME, *incrementvalue* is ignored.

Description

The lgmode structure specifies data logging options. The *mode* variable accepts either the numeric values 0 or 1 or their equivalent constants LGMOD_TIME or LGMOD_VALUE from xpcapiconst.h.

See Also

API functions xPCSetLogMode, xPCGetLogMode

scopedata

Purpose

Type definition for scope data structure

Prototype

```
typedef struct {
   int    number;
   int    type;
   int    state;
   int    signals[10];
   int    numsamples;
   int    decimation;
   int    triggermode;
   int    numprepostsamples;
   int    triggersignal
   int    triggerscope;
   int    triggerscopesample;
   double triggerlevel;
   int    triggerslope;
} scopedata;
```

Arguments

number	The scope number.
type	Determines whether the scope is displayed on the host computer or on the target computer. Values are one of the following: 1 Host 2 Target
state	Indicates the scope state. Values are one of the following: 0 Waiting to start 1 Scope is waiting for a trigger 2 Data is being acquired 3 Acquisition is finished 4 Scope is stopped (interrupted)

	5 Scope is preacquiring data
signals	List of signal indices from the target object to display on the scope.
numsamples	Number of contiguous samples captured during the acquisition of a data package.
decimation	A number, N, meaning every Nth sample is acquired in a scope window.
triggermode	Trigger mode for a scope. Values are one of the following: 0 FreeRun (default) 1 Software 2 Signal 3 Scope
numprepostsamples	If this value is less than 0, this is the number of samples to be saved before a trigger event. If this value is greater than 0, this is the number of samples to skip after the trigger event before data acquisition begins.
triggersignal	If *triggermode* = 2 for signal, identifies the block output signal to use for triggering the scope. You identify the signal with a signal index.
triggerscope	If *triggermode* = 3 for scope, identifies the scope to use for a trigger. A scope can be set to trigger when another scope is triggered.
triggerscopesample	If *triggermode* = 3 for scope, specifies the number of samples to be acquired by the triggering scope before triggering a second scope. This must be a nonnegative value.

triggerlevel	If *triggermode* = 2 for 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.
triggerslope	If *triggermode* = 2 for signal, indicates whether the trigger is on a rising or falling signal. Values are 0 Either rising or falling (default) 1 Rising 2 Falling

Description

The `scopedata` structure holds the data about a scope used in the functions `xPCGetScope` and `xPCSetScope`. In the structure, the fields are as in the various `xPCGetSc*` functions (for example, *state* is as in `xPCScGetState`, *signals* is as in `xPCScGetSignals`, etc.). The signal vector is an array of the signal identifiers, terminated by -1.

See Also

API functions `xPCSetScope`, `xPCGetScope`, `xPCScGetType`, `xPCScGetState`, `xPCScGetSignals`, `xPCScGetNumSamples`, `xPCScGetDecimation`, `xPCScGetTriggerMode`, `xPCScGetNumPrePostSamples`, `xPCScGetTriggerSignal`, `xPCScGetTriggerScope`, `xPCScGetTriggerLevel`, `xPCScGetTriggerSlope`

Purpose

Create new scope

Prototype

```
void xPCAddScope(int port, int scType, int scNum);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.
scType	Enter the type of scope.
scNum	Enter a number for a new scope. Values are 1, 2, 3. . .

Description

The xPCAddScope function creates a new scope on the target PC. For *scType*, scopes can be of type host or target, depending on the value of *scType*:

- SCTYPE_HOST for type host
- SCTYPE_TARGET for type target
- SCTYPE_FILE for type file

Constants for *scType* are defined in the header file xpcapiconst.h as SCTYPE_HOST, SCTYPE_TARGET, and SCTYPE_FILE.

Calling the xPCAddScope function with *scNum* having the number of an existing scope produces an error. Use xPCGetScopes to find the numbers of existing scopes.

See Also

API functions xPCScAddSignal, xPCScRemSignal, xPCRemScope, xPCSetScope, xPCGetScope, xPCGetScopes

Target object method addscope

xPCAverageTET

Purpose

Return average task execution time

Prototype

```
double xPCAverageTET(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCAverageTET` function returns the average task execution time (TET) for the target application.

Description

The `xPCAverageTET` function returns the TET for the target application. You can use this function when the target application is running or when it is stopped.

See Also

API functions `xPCMaximumTET`, `xPCMinimumTET`

Target object property `AvgTET`

Purpose

Close RS-232 or TCP/IP communication connection

Prototype

```
void xPCCloseConnection(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Description

The `xPCCloseConnection` function closes the RS-232 or TCP/IP communication channel opened by `xPCOpenSerialPort`, `xPCOpenTcpIpPort`, or `xPCOpenConnection`. Unlike `xPCClosePort`, it preserves the connection information such that a subsequent call to `xPCOpenConnection` succeeds without the need to resupply communication data such as the IP address or port number. To completely close the communication channel, call `xPCDeRegisterTarget`. Calling the `xPCCloseConnection` function followed by calling `xPCDeRegisterTarget` is equivalent to calling `xPCClosePort`.

See Also

API functions `xPCOpenConnection`, `xPCOpenSerialPort`, `xPCOpenTcpIpPort`, `xPCReOpenPort`, `xPCRegisterTarget`, `xPCDeRegisterTarget`

xPCClosePort

Purpose Close RS-232 or TCP/IP communication connection

Prototype

```
void xPCClosePort(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Description The `xPCClosePort` function closes the RS-232 or TCP/IP communication channel opened by either `xPCOpenSerialPort` or by `xPCOpenTcpIpPort`. Calling this function is equivalent to calling `xPCCloseConnection` and `xPCDeRegisterTarget`.

See Also API functions `xPCOpenSerialPort`, `xPCOpenTcpIpPort`, `xPCReOpenPort`, `xPCOpenConnection`, `xPCCloseConnection`, `xPCRegisterTarget`, `xPCDeRegisterTarget`

Target object method `close`

Purpose

Delete target communication properties from xPC Target API library

Prototype

```
void xPCDeRegisterTarget(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Description

The `xPCDeRegisterTarget` function causes the xPC Target API library to completely "forget" about the target communication properties. It works similarly to `xPCClosePort`, but does not close the connection to the target machine. Before calling this function, you must first call the function `xPCCloseConnection` to close the connection to the target machine. The combination of calling the `xPCCloseConnection` and `xPCDeRegisterTarget` functions has the same effect as calling `xPCClosePort`.

See Also

API functions `xPCRegisterTarget`, `xPCOpenTcpIpPort`, `xPCOpenSerialPort`, `xPCClosePort`, `xPCReOpenPort`, `xPCOpenConnection`, `xPCCloseConnection`, `xPCTargetPing`

xPCErrorMsg

Purpose Return text description for error message

Prototype

```
char *xPCErrorMsg(int error_number, char *error_message);
```

Arguments

error_number	Enter the constant of an error.
error_message	The xPCErrorMsg function copies the error message string into the buffer pointed to by *error_message*. *error_message* is then returned. You can later use *error_message* in a function such as printf. If *error_message* is NULL, the xPCErrorMsg function returns a pointer to a statically allocated string.

Return The xPCErrorMsg function returns a string associated with the error *error_number*.

Description The xPCErrorMsg function returns *error_message*, which makes it convenient to use in a printf or similar statement. Use the xPCGetLastError function to get the constant for which you are getting the message.

See Also API functions xPCSetLastError, xPCGetLastError

Purpose Change current directory on target PC to specified path

Prototype

```
long CD(BSTR dir);
```

Member Of `XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`dir`*	Enter the path on the target PC to change to.

Return If there is an error, this method returns `-1`. Otherwise, the method returns `0`.

Description The `xPCFileSystem.CD` method changes the current directory on the target PC to the path specified in *`dir`*. Use the `xPCFileSystem.PWD` method to show the current directory of the target PC.

See Also API method `xPCFileSystem.PWD`

Purpose

Close file on target PC

Prototype

`CloseFile(long filehandle);`

Member Of

`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in] filehandle`	Enter the file handle of an open file on the target PC.

Return

If there is an error, this method returns `-1`. Otherwise, the method returns `0`.

Description

The `xPCFileSystem.CloseFile` method closes the file associated with *`fileHandle`* on the target PC. *`fileHandle`* is the handle of a file previously opened by the `xPCFileSystem.OpenFile` method.

See Also

API methods `xPCFileSystem.OpenFile`, `xPCFileSystem.ReadFile`, `xPCFileSystem.WriteFile`

Purpose

Return contents of target PC directory

Prototype

`DirList(BSTR path);`

Member Of

`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`path`*	Enter the path of the directory.

Description

The `xPCFileSystem.DirList` method returns the contents of the target PC directory specified by *`path`* as an array of the `FSDir` structure.

See Also

API structure `FSDir`

API method `xPCFileSystem.GetDiskInfo`

xPCFileSystem.GetDiskInfo

Purpose

Return disk information

Prototype

`GetDiskInfo(BSTR` *`driveLetter`*`);`

Member Of

`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`driveLetter`*	Enter the driver letter that contains the file system.

Description

The `xPCFileSystem.GetDiskInfo` method accepts as input the drive specified by *`driveLetter`* and fills in the fields of the `FSDiskInfo` structure.

See Also

API structure `FSDiskInfo`

API method `xPCFileSystem.DirList`

Purpose

Return size of file on target PC

Prototype

`long GetFileSize(long filehandle);`

Member Of

`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`filehandle`*	Enter the file handle of an open file on the target PC.

Return

This method returns the size of the specified file in bytes.

Description

The `xPCFileSystem.GetFileSize` method returns the size, in bytes, of the file associated with *`filehandle`* on the target PC. *`filehandle`* is the handle of a file previously opened by the `xPCFileSystem.OpenFile` method.

See Also

API methods `xPCFileSystem.OpenFile`, `xPCFileSystem.ReadFile`

xPCFileSystem.Init

Purpose Initialize file system object to communicate with target PC

Prototype `long Init(IxPCProtocol* xPCProtocol);`

Member Of `XPCAPICOMLib.xPCFileSystem`

Arguments

[in] `xPCProtocol`	Specify the communication port of the target PC object for which the file system is to be initialized.

Return If there is an error, this method returns `-1`. Otherwise, the `xPCFileSystem.Init` method returns `0`.

Description The `xPCFileSystem.Init` method initializes the file system object to communicate with the target PC referenced by the `xPCProtocol` object.

Purpose
Create directory on target PC

Prototype
`long MKDIR(BSTR dirname);`

Member Of
`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`dirname`*	Enter the name of the directory to create on the target PC.

Return
If there is an error, this method returns `-1`. Otherwise, the method returns `0`.

Description
The `xPCFileSystem.MKDIR` method creates the directory *`dirname`* in the current directory of the target PC.

See Also
API method `xPCFileSystem.PWD`

xPCFileSystem.OpenFile

Purpose Open file on target PC

Prototype `long OpenFile(BSTR filename, BSTR permission);`

Member Of `XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`filename`*	Enter the name of the file to open on the target PC.
`[in]` *`permission`*	Enter the read/write permission with which to open the file. Values are `r` (read) or `w` (read/write).

Return The `xPCFileSystem.OpenFile` method returns the file handle for the opened file.

Description The `xPCFileSystem.OpenFile` method opens the specified file, *`filename`*, on the target PC. If the file does not exist, the `xPCFileSystem.OpenFile` method creates *`filename`*, then opens it. You can open a file for read or read/write access.

See Also API methods `xPCFileSystem.CloseFile`, `xPCFileSystem.GetFileSize`, `xPCFileSystem.ReadFile`, `xPCFileSystem.WriteFile`

Purpose Get current directory of target PC

Prototype `BSTR PWD();`

Member Of `XPCAPICOMLib.xPCFileSystem`

Return This method returns the path of the current directory on the target PC.

Description The `xPCFileSystem.PWD` method places the path of the current directory on the target PC.

See Also API method `xPCFileSystem.CD`

xPCFileSystem.ReadFile

Purpose

Read open file on target PC

Prototype

`VARIANT ReadFile(int *fileHandle*, int *start*, int *numbytes*);`

Member Of

`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`fileHandle`*	Enter the file handle of an open file on the target PC.
`[in]` *`start`*	Enter an offset from the beginning of the file from which this method can start to read.
`[in]` *`numbytes`*	Enter the number of bytes this method is to read from the file.

Return

This method returns the results of the read operation as a `VARIANT` of type `Byte`. If there is an error, this method returns `VT_ERROR`, whose value is `10`, instead.

Description

The `xPCFileSystem.ReadFile` method reads an open file on the target PC and returns the results of the read operation as a `VARIANT` of type `Byte`. *`fileHandle`* is the file handle of a file previously opened by `xPCFileSystem.OpenFile`. You can specify that the read operation begin at the beginning of the file (default) or at a certain offset into the file (*`start`*). The *`numbytes`* parameter specifies how many bytes the `xPCFileSystem.ReadFile` method is to read from the file.

See Also

API methods `xPCFileSystem.CloseFile`, `xPCFileSystem.GetFileSize`, `xPCFileSystem.OpenFile`, `xPCFileSystem.WriteFile`

Purpose

Remove file from target PC

Prototype

```
long RemoveFile(BSTR filename);
```

Member Of

`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`filename`*	Enter the name of a file on the target PC.

Return

If there is an error, this method returns `-1`. Otherwise, the method returns `0`.

Description

The `xPCFileSystem.RemoveFile` method removes the file named *`filename`* from the target PC file system. *`filename`* can be a relative or absolute pathname on the target PC.

xPCFileSystem.RMDIR

Purpose

Remove directory from target PC

Prototype

`long RMDIR(BSTR dirname);`

Member Of

`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`dirname`*	Enter the name of a directory on the target PC.

Return

If there is an error, this method returns `-1`. Otherwise, the method returns `0`.

Description

The `xPCFileSystem.RMDIR` method removes a directory named *`dirname`* from the target PC file system. *`dirname`* can be a relative or absolute pathname on the target PC.

Purpose
Get name of file for scope

Prototype
`BSTR ScGetFileName(long *scNum*);`

Member Of
`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`scNum`*	Enter the scope number.

Return
Returns the name of the file for the scope.

Description
The `xPCFileSystem.ScGetFileName` method returns the name of the file to which scope *`scNum`* will save signal data.

See Also
API method `xPCFileSystem.ScSetFileName`

xPCFileSystem.ScGetWriteMode

Purpose Get write mode of file for scope

Prototype `long ScGetWriteMode(long scNum);`

Member Of `XPCAPICOMLib.xPCFileSystem`

Arguments

`[in] scNum`	Enter the scope number.

Return This method returns the number indicating the write mode. Values are

0	Lazy mode. The FAT entry is updated only when the file is closed and not during each file write operation. This mode is faster, but if the system crashes before the file is closed, the file system might not have the actual file size (the file contents, however, will be intact).
1	Commit mode. Each file write operation simultaneously updates the FAT entry for the file. This mode is slower, but the file system always has the actual file size.

Description The `xPCFileSystem.ScGetWriteMode` method returns the write mode of the file for the scope.

See Also API method `xPCFileSystem.ScSetWriteMode`

Purpose Get block write size of data chunks

Prototype `long ScGetWriteSize(long scNum);`

Member Of `XPCAPICOMLib.xPCFileSystem`

Arguments

`[in] scNum`	Enter the scope number.

Return This method returns the block size, in bytes, of the data chunks.

Description The `xPCFileSystem.ScGetWriteSize` method gets the block size, in bytes, of the data chunks.

See Also API method `xPCFileSystem.ScSetWriteSize`

xPCFileSystem.ScSetFileName

Purpose Specify file name to contain signal data

Prototype `long ScSetFileName(long scNum, BSTR filename);`

Member Of `XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *scNum*	Enter the scope number.
`[in]` *filename*	Enter the name of a file to contain the signal data.

Return If there is an error, this method returns `-1`. Otherwise, the method returns `0`.

Description The `xPCFileSystem.ScSetFileName` method sets the name of the file to which the scope will save the signal data. xPC Target creates this file in the target PC file system. Note that you can only call this method when the scope is stopped.

See Also API method `xPCFileSystem.ScGetFileName`

Purpose

Specify when file allocation table entry is updated

Prototype

`long ScSetWriteMode(long scNum, long writeMode);`

Member Of

`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in] scNum`	Enter the scope number.	
`[in] writeMode`	Enter an integer for the write mode:	
	`0`	Enables lazy write mode
	`1`	Enables commit write mode

Return

If there is an error, this method returns `-1`. Otherwise, the method returns `0`.

Description

The `xPCFileSystem.ScSetWriteMode` method specifies when a file allocation table (FAT) entry is updated. Both modes write the signal data to the file, as follows:

`0`	Lazy mode. The FAT entry is updated only when the file is closed and not during each file write operation. This mode is faster, but if the system crashes before the file is closed, the file system might not have the actual file size (the file contents, however, will be intact).
`1`	Commit mode. Each file write operation simultaneously updates the FAT entry for the file. This mode is slower, but the file system always has the actual file size.

See Also

API method `xPCFileSystem.ScSetWriteMode`

Scope object property `Mode`

xPCFileSystem.ScSetWriteSize

Purpose

Specify that memory buffer collect data in multiples of write size

Prototype

```
long ScSetWriteSize(long scNum, long writeSize);
```

Member Of

`XPCAPICOMLib.xPCFileSystem`

Arguments

`[in]` *`scNum`*	Enter the scope number.
`[in]` *`writeSize`*	Enter the block size, in bytes, of the data chunks.

Return

If there is an error, this method returns `-1`. Otherwise, the method returns `0`.

Description

The `xPCFileSystem.ScSetWriteSize` method specifies that a memory buffer collect data in multiples of *`writeSize`*. By default, this parameter is 512 bytes, which is the typical disk sector size. Using a block size that is the same as the disk sector size provides optimal performance. *`writeSize`* must be a multiple of 512.

See Also

API method `xPCFileSystem.ScGetWriteSize`

Scope object property `WriteSize`

Purpose

Write to file on target PC

Prototype

```
long WriteFile(long fileHandle, long numbytes,
VARIANT buffer);
```

Member Of

`XPCAPICOMLib.xPCFileSystem`

Arguments

[in] *fileHandle*	Enter the file handle of an open file on the target PC.
[in] *numbytes*	Enter the number of bytes this method is to write into the file.
[in] *buffer*	The contents to write to *fileHandle* are stored in *buffer*.

Return

If there is an error, this method returns `-1`. Otherwise, the method returns `0`.

Description

The `xPCFileSystem.WriteFile` method writes the contents of the `VARIANT` *buffer*, of type `Byte`, to the file specified by *fileHandle* on the target PC. The *fileHandle* parameter is the handle of a file previously opened by `xPCFSOpenFile`. *numbytes* is the number of bytes to write to the file.

See Also

API methods `xPCFileSystem.CloseFile`, `xPCFileSystem.GetFileSize`, `xPCFileSystem.OpenFile`, `xPCFileSystem.ReadFile`

xPCFreeAPI

Purpose Unload xPC Target DLL

Prototype

```
int xPCFreeAPI(void);
```

Arguments none

Description The `xPCFreeAPI` function unloads the xPC Target dynamic link library. You must execute this function once at the end of the application to unload the xPC Target API DLL. This frees the memory allocated to the functions. This function is defined in the file `xpcinitfree.c`. Link this file with your application.

See Also API functions `xPCInitAPI`, `xPCNumLogWraps`, `xPCNumLogSamples`, `xPCMaxLogSamples`, `xPCGetStateLog`, `xPCGetTETLog`, `xPCSetLogMode`, `xPCGetLogMode`

xPCFSCD

Purpose

Change current directory on target PC to specified path

Prototype

```
void xPCFSCD(int port, char *dir);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
dir	Enter the path on the target PC to change to.

Description

The `xPCFSCD` function changes the current directory on the target PC to the path specified in *`dir`*. Use the `xPCFSGetPWD` function to show the current directory of the target PC.

See Also

API function `xPCFSGetPWD`

File object method `cd`

xPCFSCloseFile

Purpose

Close file on target PC

Prototype

```
void xPCFSCloseFile(int port, int fileHandle);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
fileHandle	Enter the file handle of an open file on the target PC.

Description

The `xPCFSCloseFile` function closes the file associated with *fileHandle* on the target PC. *fileHandle* is the handle of a file previously opened by the `xPCFSOpenFile` function.

See Also

API functions `xPCFSOpenFile`, `xPCFSReadFile`, `xPCFSWriteFile`

File object method `fclose`

Purpose

Get contents of specified directory on target PC

Prototype

```
void xPCFSDir(int port, const char *path, char
*data, int numbytes);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
path	Enter the path on the target PC.
data	The contents of the directory are stored in *data*, whose allocated size is specified in *numbytes*.
numbytes	Enter the size, in bytes, of the array *data*.

Description

The `xPCFSDir` function copies the contents of the target PC directory specified by *path* into `data`. The `xPCFSDir` function returns the listing in the *data* array, which must be of size *numbytes*. Use the `xPCFSDirSize` function to obtain the size of the directory for the *numbytes* parameter.

See Also

API function `xPCFSDirSize`

File object method `dir`

xPCFSDirItems

Purpose

Get contents of specified directory on target PC

Prototype

```
void xPCFSDirItems(int port, const char *path, dirStruct
*dirs, int numDirItems);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
path	Enter the path on the target PC.
dirs	Enter the structure to contain the contents of the directory.
numDirItems	Enter the number of items in the directory.

Description

The `xPCFSDirItems` function copies the contents of the target PC directory specified by *path*. The `xPCFSDirItems` function copies the listing into the *dirs* structure, which must be of size *numDirItems*. Use the `xPCFSDirStructSize` function to obtain the size of the directory for the *numDirItems* parameter.

See Also

API functions `xPCFSDirStructSize`, `dirStruct`

File object method `dir`

xPCFSDirSize

Purpose

Return size of specified directory on target PC

Prototype

`int xPCFSDirSize(int *port*, const char **path*);`

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
path	Enter the directory path on the target PC.

Return

The `xPCFSDirSize` function returns the size, in bytes, of the specified directory.

Description

The `xPCFSDirSize` function returns the size, in bytes, of the buffer needed to get the directory listing of the directory on the target PC. Use this size as the *`numbytes`* parameter in the `xPCFSDir` function.

See Also

API function `xPCFSDirItems`

File object method `dir`

xPCFSDirStructSize

Purpose

Get number of items in directory

Prototype

```
int xPCFSDirSize(int port, const char *path);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
path	Enter the directory path on the target PC.

Description

The `xPCFSDirStructSize` function returns the number of items in the directory on the target PC. Use this size as the *`numDirItems`* parameter in the `xPCFSDirItems` function.

See Also

API function `xPCFSDir`

File object method `dir`

Purpose

Information about target PC file system

Prototype

```
diskinfo xPCFSDiskInfo(int  port, const char *driveletter);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
driveletter	Enter the drive letter of the file system for which you want information.

Description

The `xPCFSDiskInfo` function returns disk information for the file system of the specified target PC drive, *driveletter*. This function returns this information in the `diskinfo` structure.

See Also

API structure `diskinfo`

xPCFSGetError

Purpose Get text description for error number on target PC file system

Prototype

```
void xPCFSGetError(int port, unsigned int error_number,
char *error_message);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
error_number	Enter the constant of an error.
error_message	The string of the message associated with the error *error_number* is stored in *error_message*.

Description The `xPCFSGetError` function gets the *error_message* associated with *error_number*. This enables you to use the error message in a `printf` or similar statement.

Purpose Return size of file on target PC

Prototype

```
int xPCFSGetFileSize(int port, int fileHandle);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
fileHandle	Enter the file handle of an open file on the target PC.

Return Return the size of the specified file in bytes.

Description The `xPCFSGetFileSize` function returns the size, in bytes, of the file associated with *fileHandle* on the target PC. *fileHandle* is the handle of a file previously opened by the `xPCFSOpenFile` function.

See Also API functions `xPCFSOpenFile`, `xPCFSReadFile`

File object methods `fopen`, `fread`

xPCFSGetPWD

Purpose

Get current directory of target PC

Prototype

```
void xPCFSGetPWD(int port, char *pwd);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
pwd	The path of the current directory is stored in *pwd*.

Description

The `xPCFSGetPWD` function places the path of the current directory on the target PC in *pwd*, which must be allocated by the caller.

See Also

File object method `pwd`

Purpose

Create new directory on target PC

Prototype

```
void xPCFSMKDIR(int port, const char *dirname);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
dirname	Enter the name of the directory to create on the target PC.

Description

The `xPCFSMKDIR` function creates the directory *dirname* in the current directory of the target PC.

See Also

API function `xPCFSGetPWD`

File object method `mkdir`

xPCFSOpenFile

Purpose

Open file on target PC

Prototype

```
int xPCFSOpenFile(int port, const char *filename,
const char *permission);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.
filename	Enter the name of the file to open on the target PC.
permission	Enter the read/write permission with which to open the file. Values are r (read) or w (read/write).

Return

The xPCFSOpenFile function returns the file handle for the opened file. If there is an error, this function returns -1.

Description

The xPCFSOpenFile function opens the specified file, *filename*, on the target PC. If the file does not exist, the xPCFSOpenFile function creates *filename*, then opens it. You can open a file for read or read/write access.

See Also

API functions xPCFSCloseFile, xPCFSGetFileSize, xPCFSReadFile, xPCFSWriteFile

File object methods fclose, filetable, fopen, fread, fwrite

Purpose

Read open file on target PC

Prototype

```
void xPCFSReadFile(int port, int fileHandle, int start,
int numbytes, unsigned char *data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
fileHandle	Enter the file handle of an open file on the target PC.
start	Enter an offset from the beginning of the file from which this function can start to read.
numbytes	Enter the number of bytes this function is to read from the file.
data	The contents of the file are stored in *data*.

Description

The `xPCFSReadFile` function reads an open file on the target PC and places the results of the read operation in the array *data*. *fileHandle* is the file handle of a file previously opened by `xPCFSOpenFile`. You can specify that the read operation begin at the beginning of the file (default) or at a certain offset into the file (*start*). The *numbytes* parameter specifies how many bytes the `xPCFSReadFile` function is to read from the file.

See Also

API functions `xPCFSCloseFile`, `xPCFSGetFileSize`, `xPCFSOpenFile`, `xPCFSWriteFile`

File object methods `fopen`, `fread`

xPCFSRemoveFile

Purpose

Remove file from target PC

Prototype

```
void xPCFSRemoveFile(int port, const char *filename);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
filename	Enter the name of a file on the target PC.

Description

The `xPCFSRemoveFile` function removes the file named *filename* from the target PC file system. *filename* can be a relative or absolute pathname on the target PC.

See Also

File object method `removefile`

Purpose Remove directory from target PC

Prototype

```
void xPCFSRMDIR(int port, const char *dirname);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
dirname	Enter the name of a directory on the target PC.

Description The `xPCFSRMDIR` function removes a directory named *dirname* from the target PC file system. *dirname* can be a relative or absolute pathname on the target PC.

See Also File object method `rmdir`

xPCFSScGetFilename

Purpose

Get name of file for scope

Prototype

```
const char *xPCFSScGetFilename(int port, int
scNum, char *filename);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
filename	The name of the file for the specified scope is stored in *filename*.

Return

Returns the value of *filename*, the name of the file for the scope.

Description

The `xPCFSScGetFilename` function returns the name of the file to which scope *scNum* will save signal data. *filename* points to a caller-allocated character array to which the filename is copied.

See Also

API function `xPCFSScSetFilename`

Scope object property `Filename`

Purpose

Get write mode of file for scope

Prototype

```
int xPCFSScGetWriteMode(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

Returns the number indicating the write mode. Values are

0	Lazy mode. The FAT entry is updated only when the file is closed and not during each file write operation. This mode is faster, but if the system crashes before the file is closed, the file system might not have the actual file size (the file contents, however, will be intact).
1	Commit mode. Each file write operation simultaneously updates the FAT entry for the file. This mode is slower, but the file system always has the actual file size.

Description

The `xPCFSScGetWriteMode` function returns the write mode of the file for the scope.

See Also

API function `xPCFSScSetWriteMode`

Scope object property `Mode`

xPCFSScGetWriteSize

Purpose

Get block write size of data chunks

Prototype

```
unsigned int xPCFSScGetWriteSize(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

Returns the block size, in bytes, of the data chunks.

Description

The `xPCFSScGetWriteSize` function gets the block size, in bytes, of the data chunks.

See Also

API function `xPCFSScSetWriteSize`

Scope object property `WriteSize`

Purpose Specify name for file to contain signal data

Prototype

```
void xPCFSScSetFilename(int port, int scNum,
const char *filename);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
filename	Enter the name of a file to contain the signal data.

Description The `xPCFSScSetFilename` function sets the name of the file to which the scope will save the signal data. xPC Target creates this file in the target PC file system. Note that you can only call this function when the scope is stopped.

See Also API function `xPCFSScGetFilename`

Scope object property `Filename`

xPCFSScSetWriteMode

Purpose

Specify when file allocation table entry is updated

Prototype

```
void xPCFSScSetWriteMode(int port, int scNum, int writeMode);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
writeMode	Enter an integer for the write mode: `0` Enables lazy write mode `1` Enables commit write mode

Description

The `xPCFSScSetWriteMode` function specifies when a file allocation table (FAT) entry is updated. Both modes write the signal data to the file, as follows:

`0`	Lazy mode. The FAT entry is updated only when the file is closed and not during each file write operation. This mode is faster, but if the system crashes before the file is closed, the file system might not have the actual file size (the file contents, however, will be intact).
`1`	Commit mode. Each file write operation simultaneously updates the FAT entry for the file. This mode is slower, but the file system always has the actual file size.

See Also

API function `xPCFSScGetWriteMode`

Scope object property `Mode`

Purpose

Specify that memory buffer collect data in multiples of write size

Prototype

```
void xPCFSScSetWriteSize(int port, int scNum, unsigned int
writeSize);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
writeSize	Enter the block size, in bytes, of the data chunks.

Description

The `xPCFSScSetWriteSize` function specifies that a memory buffer collect data in multiples of *writeSize*. By default, this parameter is 512 bytes, which is the typical disk sector size. Using a block size that is the same as the disk sector size provides optimal performance. *writeSize* must be a multiple of 512.

See Also

API function `xPCFSScGetWriteSize`

Scope object property `WriteSize`

xPCFSWriteFile

Purpose

Write to file on target PC

Prototype

```
void xPCFSWriteFile(int port, int fileHandle, int numbytes,
const unsigned char *data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
fileHandle	Enter the file handle of an open file on the target PC.
numbytes	Enter the number of bytes this function is to write into the file.
data	The contents to write to *fileHandle* are stored in *data*.

Description

The `xPCFSWriteFile` function writes the contents of the array *data* to the file specified by *fileHandle* on the target PC. The *fileHandle* parameter is the handle of a file previously opened by `xPCFSOpenFile`. *numbytes* is the number of bytes to write to the file.

See Also

API functions `xPCFSCloseFile`, `xPCFSGetFileSize`, `xPCFSOpenFile`, `xPCFSReadFile`

Purpose Get version number of xPC Target API

Prototype

```
const char *xPCGetAPIVersion(void);
```

Arguments none

Return The `xPCGetApiVersion` function returns a string with the version number of the xPC Target kernel on the target PC.

Description The `xPCGetApiVersion` function returns a string with the version number of the xPC Target kernel on the target PC. The string is a constant string within the API DLL. Do not modify this string.

See Also API function `xPCGetTargetVersion`

xPCGetAppName

Purpose

Return target application name

Prototype

`char *xPCGetAppName(int port, char *model_name);`

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
model_name	The `xPCGetAppName` function copies the target application name string into the buffer pointed to by *model_name*. *model_name* is then returned. You can later use *model_name* in a function such as `printf`. Note that the maximum size of the buffer is 256 bytes. To ensure that you have enough space for the application name string, allocate a buffer of size 256 bytes.

Return

The `xPCGetAppName` function returns a string with the name of the target application.

Description

The `xPCGetAppName` function returns the name of the target application. You can use the return value, *model_name*, in a `printf` or similar statement. In case of error, the name string is unchanged.

Examples

Allocate 256 bytes for the buffer `appname`.

```
char *appname=malloc(256);
xPCGetAppName(iport,appname);
appname=realloc(appname,strlen(appname)+1);
...
free(appname);
```

See Also

API function `xPCIsAppRunning`

Target object property `Application`

xPCGetEcho

Purpose

Return display mode for target message window

Prototype

```
int xPCGetEcho(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCGetEcho` function returns the number indicating the display mode. Values are

1	Display is on. Messages are displayed in the message display window on the target.
0	Display is off.

Description

The `xPCGetEcho` function returns the display mode of the target PC using communication channel *port*. Messages include the status of downloading the target application, changes to parameters, and changes to scope signals.

See Also

API function `xPCSetEcho`

xPCGetExecTime

Purpose

Return target application execution time

Prototype

```
double xPCGetExecTime(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCGetExecTime` function returns the current execution time for a target application.

Description

The `xPCGetExecTime` function returns the current execution time for the running target application. If the target application is stopped, the value is the last running time when the target application was stopped. If the target application is running, the value is the current running time.

See Also

API functions `xPCSetStopTime`, `xPCGetStopTime`

Target object property `ExecTime`

Purpose Return constant of last error

Prototype `int xPCGetLastError(void);`

Return The `xPCGetLastError` function returns the error constant for the last reported error. If there is no error, this function returns 0.

Description The `xPCGetLastError` function returns the constant of the last reported error by another API function. This value is reset every time you call a new function. Therefore, you should check this constant value immediately after a call to an API function. For a list of error constants and messages, see Appendix A, "xPC Target C API Error Messages".

See Also API functions `xPCErrorMsg`, `xPCSetLastError`

xPCGetLoadTimeOut

Purpose

Return initialization time-out value of target application

Prototype

```
int xPCGetLoadTimeOut(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCGetLoadTimeOut` function returns the number of seconds allowed for the initialization of the target application. If there is an error, this function returns `-1`.

Description

The `xPCGetLoadTimeOut` function returns the number of seconds allowed for the initialization of the target application.

When you load a new target application onto the target PC, the function `xPCLoadApp` waits for a certain amount of time before checking to see if the initialization of the target application is complete. In the case where initialization of the target application is not complete, the function `xPCLoadApp` returns a time-out error. By default, `xPCLoadApp` checks five times to see whether the target application is ready, with each attempt taking about 1 second. However, in the case of larger models or models requiring longer initialization (for example, those with thermocouple boards), the default of about 5 seconds might not be sufficient and a spurious time-out is generated. The function `xPCSetLoadTimeOut` sets the time-out to a different number.

Use the `xPCGetLoadTimeOut` function if you suspect that the current number of seconds (the time-out value) is too short. Then use the `xPCSetLoadTimeOut` function to set the time-out to a higher number.

See Also

API functions `xPCLoadApp`, `xPCUnloadApp`, `xPCSetLoadTimeOut`

"Increasing the Time-Out Value" in the Getting Started with xPC Target documentation.

Purpose

Return logging mode and increment value for target application

Prototype

```
lgmode xPCGetLogMode(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCGetLogMode` function returns the logging mode in the `lgmode` structure. If the logging mode is `1` (`LGMOD_VALUE`), this function also returns an increment value in the `lgmode` structure. If an error occurs, this function returns `-1`.

Description

The `xPCGetLogMode` function gets the logging mode and increment value for the current target application. The increment (difference in amplitude) value is measured between logged data points. A data point is logged only when an output signal or a state changes by the increment value.

See Also

API function `xPCSetLogMode`

API structure `lgmode`

xPCGetNumOutputs

Purpose

Return number of outputs

Prototype

```
int xPCGetNumOutputs(int port);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.

Return

The xPCGetNumOutputs function returns the number of outputs in the current target application.

Description

The xPCGetNumOutputs function returns the number of outputs in the target application. The number of outputs equals the sum of the input signal widths of all output blocks at the root level of the Simulink model.

See Also

API functions xPCGetOutputLog, xPCGetNumStates, xPCGetStateLog

Purpose

Return number of tunable parameters

Prototype

```
int xPCGetNumParams(int port);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.

Return

The xPCGetNumParams function returns the number of tunable parameters in the target application.

Description

The xPCGetNumParams function returns the number of tunable parameters in the target application. Use this function to see how many parameters you can get or modify.

See Also

API functions xPCGetParamIdx, xPCSetParam, xPCGetParam, xPCGetParamName, xPCGetParamDims

Target object property NumParameters

xPCGetNumSignals

Purpose

Return number of signals

Prototype

```
int xPCGetNumSignals(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCGetNumSignals` function returns the number of signals in the target application.

Description

The `xPCGetNumSignals` function returns the total number of signals in the target application that can be monitored from the host. Use this function to see how many signals you can monitor.

See Also

API functions `xPCGetSignalIdx`, `xPCGetSignal`, `xPCGetSignals`, `xPCGetSignalName`, `xPCGetSignalWidth`

Target object property `NumSignals`

Purpose

Return number of states

Prototype

```
int xPCGetNumStates(int port);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.

Return

The xPCGetNumStates function returns the number of states in the target application.

Description

The xPCGetNumStates function returns the number of states in the target application.

See Also

API functions xPCGetStateLog, xPCGetNumOutputs, xPCGetOutputLog

Target object property StateLog

xPCGetOutputLog

Purpose

Copy output log data to array

Prototype

```
void xPCGetOutputLog(int port, int first_sample,
int num_samples,
int decimation, int output_id, double *output_data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
first_sample	Enter the index of the first sample to copy.
num_samples	Enter the number of samples to copy from the output log.
decimation	Select whether to copy all the sample values or every `N`th value.
output_id	Enter an output identification number.
output_data	The log is stored in *output_data*, whose allocation is the responsibility of the caller.

Description

The `xPCGetOutputLog` function gets the output log and copies that log to an array. You get the data for each output signal in turn by specifying *output_id*. Output IDs range from `0` to (`N`-1), where `N` is the return value of `xPCGetNumOutputs`. Entering `1` for *decimation* copies all values. Entering `N` copies every `N`th value.

For *first_sample*, the sample indices range from `0` to (`N`-1), where `N` is the return value of `xPCNumLogSamples`. Get the maximum number of samples by calling the function `xPCNumLogSamples`.

Note that the target application must be stopped before you get the number.

See Also

API functions xPCNumLogWraps, xPCNumLogSamples, xPCMaxLogSamples, xPCGetNumOutputs, xPCGetStateLog, xPCGetTETLog, xPCGetTimeLog

Target object method getlog

Target object property OutputLog

xPCGetParam

Purpose Get parameter value and copy it to array

Prototype

```
void xPCGetParam(int port, int paramIndex,
double *paramValue);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
paramIndex	Enter the index for a parameter.
paramValue	The function returns a parameter value as an array of doubles.

Description The `xPCGetParam` function returns the parameter as an array in *paramValue*. *paramValue* must be of sufficient size to hold the parameter. You can query the size by calling the function `xPCGetParamDims`. Get the parameter index by calling the function `xPCGetParamIdx`. The parameter matrix is returned as a vector, with the conversion being done in column-major format. It is also returned as a double, regardless of the data type of the actual parameter.

For *paramIndex*, values range from `0` to (`N-1`), where `N` is the return value of `xPCGetNumParams`.

See Also API functions `xPCSetParam`, `xPCGetParamDims`, `xPCGetParamIdx`, `xPCGetNumParams`

Target object method `getparamid`

Target object properties `ShowParameters`, `Parameters`

xPCGetParamDims

Purpose

Get row and column dimensions of parameter

Prototype

```
void xPCGetParamDims(int port, int paramIndex,
int *dimension);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
paramIndex	Parameter index.
dimension	Dimensions (row, column) of a parameter.

Description

The `xPCGetParamDims` function gets the dimensions (row, column) of a parameter with *paramIndex* and stores them in *dimension*, which must have at least two elements.

For *paramIndex*, values range from `0` to (`N-1`), where `N` is the return value of `xPCGetNumParams`.

See Also

API functions `xPCGetParamIdx`, `xPCGetParamName`, `xPCSetParam`, `xPCGetParam`, `xPCGetNumParams`

Target object method `getparamid`

Target object properties `ShowParameters`, `Parameters`

xPCGetParamIdx

Purpose

Return parameter index

Prototype

```
int xPCGetParamIdx(int port, const char *blockName,
const char *paramName);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.
blockName	Enter the full block path generated by Real-Time Workshop.
paramName	Enter the parameter name for a parameter associated with the block.

Return

The xPCGetParamIdx function returns the parameter index for the parameter name. If there is an error, this function returns -1.

Description

The xPCGetParamIdx function returns the parameter index for the parameter name (*paramName*) associated with a Simulink block (*blockName*). Both *blockName* and *paramName* must be identical to those generated at target application building time. The block names should be referenced from the file model_namept.m in the generated code, where *model_name* is the name of the model. Note that a block can have one or more parameters.

See Also

API functions xPCGetParamDims, xPCGetParamName, xPCGetParam

Target object method getparamid

Target object properties ShowParameters, Parameters

xPCGetParamName

Purpose

Get name of parameter

Prototype

```
void xPCGetParamName(int port, int paramIdx,
char *blockName, char
*paramName);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
paramIdx	Enter a parameter index.
blockName	String with the full block path generated by Real-Time Workshop.
paramName	Name of a parameter for a specific block.

Description

The `xPCGetParamName` function gets the parameter name and block name for a parameter with the index *paramIdx*. The block path and name are returned and stored in *blockName*, and the parameter name is returned and stored in *paramName*. You must allocate sufficient space for both *blockName* and *paramName*. If the *paramIdx* is invalid, `xPCGetLastError` returns nonzero, and the strings are unchanged. Get the parameter index from the function `xPCGetParamIdx`.

See Also

API functions `xPCGetParam`, `xPCGetParamDims`, `xPCGetParamIdx`

Target object properties `ShowParameters`, `Parameters`

xPCGetSampleTime

Purpose

Return target application sample time

Prototype

```
double xPCGetSampleTime(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCGetSampleTime` function returns the sample time, in seconds, of the target application. If there is an error, this function returns `-1`.

Description

The `xPCGetSampleTime` function returns the sample time, in seconds, of the target application. You can get the error by using the function `xPCGetLastError`.

See Also

API function `xPCSetSampleTime`

Target object property `SampleTime`

Purpose

Get and copy scope data to structure

Prototype

```
scopedata xPCGetScope(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

The `xPCGetScope` function returns a structure of type `scopedata`.

Description

The `xPCGetScope` function gets properties of a scope with *scNum* and copies the properties into a structure with type `scopedata`. You can use this function in conjunction with `xPCSetScope` to change several properties of a scope at one time. See `scopedata` for a list of properties. Use the `xPCGetScope` function to get the scope number.

See Also

API functions `xPCSetScope`, `scopedata`

Target object method `getscope`

xPCGetScopes

Purpose

Get and copy list of scope numbers

Prototype

```
void xPCGetScopes(int port, int *data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
data	List of scope numbers in an integer array (allocated by the caller) as a list of unsorted integers and terminated by `-1`.

Description

The `xPCGetScopes` function gets the list of scopes currently defined. You can use the constant `MAX_SCOPES` (defined in `xpcapiconst.h`) as the size of *data*. This is currently set to 30 scopes.

See Also

API functions `xPCSetScope`, `xPCGetScope`, `xPCScGetSignals`

Target object property `Scopes`

xPCGetSignal

Purpose

Return value of signal

Prototype

```
double xPCGetSignal(int port, int sigNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
sigNum	Enter a signal number.

Return

The `xPCGetSignal` function returns the current value of signal *sigNum*.

Description

The `xPCGetSignal` function returns the current value of a signal. For vector signals, use `xPCGetSignals` rather than call this function multiple times. Use the `xPCGetSignalIdx` function to get the signal number.

See Also

API function `xPCGetSignals`

Target object properties `ShowSignals`, `Signals`

xPCGetSignalIdx

Purpose

Return index for signal

Prototype

```
int xPCGetSignalIdx(int port, const char *sigName);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
sigName	Enter a signal name.

Return

The `xPCGetSignalIdx` function returns the index for the signal with name *sigName*. If there is an error, this function returns `-1`.

Description

The `xPCGetSignalIdx` function returns the index of a signal. The name must be identical to the name generated when the application was built. You should reference the name from the file `model_namebio.m` in the generated code, where *model_name* is the name of the model. The creator of the application should already know the signal name.

See Also

API functions `xPCGetSignalName`, `xPCGetSignalWidth`, `xPCGetSignal`, `xPCGetSignals`

Target object method `getsignalid`

Purpose Return array of signal indices

Prototype

```
int xPCGetSigIdxfromLabel(int port, const char
*sigLabel, int *sigIds);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
sigLabel	String with the name of a signal label.
sigIds	Return array of signal indices.

Return The `xPCGetSigIdxfromLabel` function fills an array *sigIds* of signal indices. If no signal is found, this function returns `-1`. It returns zero (0) upon success.

Description The `xPCGetSigIdxfromLabel` function returns in *sigIds* the array of signal indices for signal *sigName*. This function assumes that you have labeled the signal for which you request the indices (see the **Signal name** parameter of the “Signal Properties Dialog Box” in the Simulink documentation). Note that xPC Target refers to Simulink signal names as signal labels. The creator of the application should already know the signal name/label.

sigIds must be large enough to contain the array of indices. You can use the `xPCGetSigLabelWidth` function to get the required amount of memory to be allocated by the `sigIds` array.

See Also API functions `xPCGetSignalLabel`, `xPCGetSigLabelWidth`

xPCGetSignalLabel

Purpose

Copy label of signal to character array

Prototype

```
char * xPCGetSignalLabel(int port, int sigIdx,
char *sigLabel);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
sigIdx	Enter signal index.
sigLabel	Return signal label associated with signal index, *sigIdx*.

Return

The `xPCGetSignalLabel` function returns the label of the signal.

Description

The `xPCGetSignalLabel` function copies and returns the signal label, including the block path, of a signal with *sigIdx*. The result is stored in *sigLabel*. If *sigIdx* is invalid, `xPCGetLastError` returns a nonzero value, and *sigLabel* is unchanged. The function returns *sigLabel*, which makes it convenient to use in a `printf` or similar statement. This function assumes that you already know the signal index.

This function assumes that you have labeled the signal for which you request the index (see the **Signal name** parameter of the "Signal Properties Dialog Box" in the Simulink documentation). Note that xPC Target refers to Simulink signal names as signal labels. The creator of the application should already know the signal name/label.

See Also

API functions `xPCGetSigIdxfromLabel`, `xPCGetSigLabelWidth`

Purpose

Return number of elements in signal

Prototype

```
int xPCGetSigLabelWidth(int port, const char *sigName);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
sigName	String with the name of a signal.

Return

The `xPCGetSigLabelWidth` function returns the number of elements that the signal `sigName` contains. If there is an error, this function returns `-1`.

Description

The `xPCGetSigLabelWidth` function returns the number of elements that the signal *sigName* contains. This function assumes that you have labeled the signal for which you request the elements (see the **Signal name** parameter of the "Signal Properties Dialog Box" in the Simulink documentation). Note that xPC Target refers to Simulink signal names as signal labels. The creator of the application should already know the signal name/label.

See Also

API functions `xPCGetSigIdxfromLabel`, `xPCGetSignalLabel`

xPCGetSignalName

Purpose

Copy name of signal to character array

Prototype

```
char *xPCGetSignalName(int port, int sigIdx,
char *sigName);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
sigIdx	Enter a signal index.
sigName	String with the name of a signal.

Return

The `xPCGetSignalName` function returns the name of the signal.

Description

The `xPCGetSignalName` function copies and returns the signal name, including the block path, of a signal with *sigIdx*. The result is stored in *sigName*. If *sigIdx* is invalid, `xPCGetLastError` returns a nonzero value, and *sigName* is unchanged. The function returns *sigName*, which makes it convenient to use in a `printf` or similar statement. This function assumes that you already know the signal index.

See Also

API functions `xPCGetSignalIdx`, `xPCGetSignalWidth`, `xPCGetSignal`, `xPCGetSignals`

Target object properties `ShowSignals`, `Signals`

Purpose

Return vector of signal values

Prototype

```
int xPCGetSignals(int port, int numSignals,
const int *signals,
double *values);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.
numSignals	Enter the number of signals to be acquired (that is, the number of values in *signals*).
signals	Enter the list of signal numbers to be acquired.
values	Returned values are stored in the double array *values*.

Return

The xPCGetSignals function returns 0 upon success. If there is an error, this function returns -1.

Description

The xPCGetSignals function is the vector version of the function xPCGetSignal. This function returns the values of a vector of signals (up to 1000) as fast as it can acquire them. The signal values are not guaranteed to be at the same time step (for that, define a scope of type SCTYPE_HOST and use xPCScGetData). xPCGetSignal does the same thing for a single signal, and could be used multiple times to achieve the same effect. However, the xPCGetSignals function is faster, and the signal values are more likely to be spaced closely together. The signals are converted to doubles regardless of the actual data type of the signal.

For *signals*, the list you provide should be stored in an integer array. Get the signal numbers with the function xPCGetSignalIdx.

See Also

API function xPCGetSignal, xPCGetSignalIdx

Example

To reference signal vector data rather than scalar values, pass a vector of indices for the signal data. For example:

```
/**********************************************************/

/* Assume a signal of width 10, with the blockpath
* mySubsys/mySignal and the signal index s1.
*/

int i;
int sigId[10];
double sigVal[10]; /* Signal values are stored here */

/* Get the ID of the first signal */
sigId[0] = xPCGetSignalIdx(port, "mySubsys/mySignal/s1");

if (sigId[0] == -1) {
/* Handle error appropriately */
}

for (i = 1; i < 10; i++) {
     sigId[i] = sigId[0] + i;
}

xPCGetSignals(port, 10, sigId, sigVal);
/* If no error, sigVal should have the signal values */

/**********************************************************/
```

To repeatedly get the signals, repeat the call to `xPCGetSignals`. If you do not change `sigID`, you only need to call `xPCGetSignalIdx` once.

xPCGetSignalWidth

Purpose

Return width of signal

Prototype

```
int xPCGetSignalWidth(int port, int sigIdx);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
sigIdx	Enter the index of a signal.

Return

The `xPCGetSignalWidth` function returns the signal width for a signal with *sigIdx*. If there is an error, this function returns `-1`.

Description

The `xPCGetSignalWidth` function returns the number of signals for a specified signal index. Although signals are manipulated as scalars, the width of the signal might be useful to reassemble the components into a vector again. A signal's width is the number of signals in the vector.

See Also

API functions `xPCGetSignalIdx`, `xPCGetSignalName`, `xPCGetSignal`, `xPCGetSignals`

xPCGetStateLog

Purpose

Copy state log values to array

Prototype

```
void xPCGetStateLog(int port, int first_sample,
int num_samples,
int decimation, int state_id, double *state_data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
first_sample	Enter the index of the first sample to copy.
num_samples	Enter the number of samples to copy from the output log.
decimation	Select whether to copy all the sample values or every `N`th value.
state_id	Enter a state identification number.
state_data	The log is stored in *state_data*, whose allocation is the responsibility of the caller.

Description

The `xPCGetStateLog` function gets the state log. It then copies the log into *state_data*. You get the data for each state signal in turn by specifying the *state_id*. State IDs range from `1` to (`N`-1), where `N` is the return value of `xPCGetNumStates`. Entering `1` for *decimation* copies all values. Entering `N` copies every `N`th value. For *first_sample*, the sample indices range from `0` to (`N`-1), where `N` is the return value of `xPCNumLogSamples`. Use the `xPCNumLogSamples` function to get the maximum number of samples.

Note that the target application must be stopped before you get the number.

See Also

API functions xPCNumLogWraps, xPCNumLogSamples, xPCMaxLogSamples, xPCGetNumStates, xPCGetOutputLog, xPCGetTETLog, xPCGetTimeLog

Target object method getlog

Target object property StateLog

xPCGetStopTime

Purpose

Return stop time

Prototype

```
double xPCGetStopTime(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCGetStopTime` function returns the stop time as a double, in seconds, of the target application. If there is an error, this function returns `-10.0`. If the stop time is infinity (run forever), this function returns `-1.0`.

Description

The `xPCGetStopTime` function returns the stop time, in seconds, of the target application. This is the amount of time the target application runs before stopping. If there is an error, this function returns `-10.0`. You will then need to use the function `xPCGetLastError` to find the error number.

See Also

API function `xPCSetStopTime`

Target object property `StopTime`

Purpose Get xPC Target kernel version

Prototype

```
void xPCGetTargetVersion(int port, char *ver);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
ver	The version is stored in *ver*.

Description The `xPCGetTargetVersion` function gets a string with the version number of the xPC Target kernel on the target PC. It then copies that version number into *ver*.

See Also `xPCGetAPIVersion`

xPCGetTETLog

Purpose

Copy TET log to array

Prototype

```
void xPCGetTETLog(int port, int first_sample,
int num_samples, int decimation,
double *TET_data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
first_sample	Enter the index of the first sample to copy.
num_samples	Enter the number of samples to copy from the TET log.
decimation	Select whether to copy all the sample values or every `N`th value.
TET_data	The log is stored in *TET_data*, whose allocation is the responsibility of the caller.

Description

The `xPCGetTETLog` function gets the task execution time (TET) log. It then copies the log into *TET_data*. Entering `1` for *decimation* copies all values. Entering `N` copies every `N`th value. For *first_sample*, the sample indices range from `0` to (`N-1`), where `N` is the return value of `xPCNumLogSamples`. Use the `xPCNumLogSamples` function to get the maximum number of samples.

Note that the target application must be stopped before you get the number.

See Also

API functions `xPCNumLogWraps`, `xPCNumLogSamples`, `xPCMaxLogSamples`, `xPCGetNumOutputs`, `xPCGetStateLog`, `xPCGetTimeLog`

Target object method `getlog`

Target object property `TETLog`

Purpose

Copy time log to array

Prototype

```
void xPCGetTimeLog(int port, int first_sample,
int num_samples,
int decimation, double *time_data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
first_sample	Enter the index of the first sample to copy.
num_samples	Enter the number of samples to copy from the time log.
decimation	Select whether to copy all the sample values or every `N`th value.
time_data	The log is stored in *time_data*, whose allocation is the responsibility of the caller.

Description

The `xPCGetTimeLog` function gets the time log and copies the log into *time_data*. This is especially relevant in the case of value-equidistant logging, where the logged values are not necessarily uniformly spaced in time. Entering `1` for *decimation* copies all values. Entering `N` copies every `N`th value. For *first_sample*, the sample indices range from `0` to (`N-1`), where `N` is the return value of `xPCNumLogSamples`. Use the `xPCNumLogSamples` function to get the number of samples.

Note that the target application must be stopped before you get the number.

See Also

API functions `xPCNumLogWraps`, `xPCNumLogSamples`, `xPCMaxLogSamples`, `xPCGetStateLog`, `xPCGetTETLog`, `xPCSetLogMode`, `xPCGetLogMode`

Target object method `getlog`

Target object property `TimeLog`

xPCInitAPI

Purpose Initialize xPC Target DLL

Prototype

```
int xPCInitAPI(void);
```

Arguments none

Return The `xPCInitAPI` function returns `0` upon success. If there is an error, this function returns `-1`.

Description The `xPCInitAPI` function initializes the xPC Target dynamic link library. You must execute this function once at the beginning of the application to load the xPC Target API DLL. This function is defined in the file `xpcinitfree.c`. Link this file with your application.

See Also API functions `xPCFreeAPI`, `xPCNumLogWraps`, `xPCNumLogSamples`, `xPCMaxLogSamples`, `xPCGetStateLog`, `xPCGetTETLog`, `xPCSetLogMode`, `xPCGetLogMode`

xPCIsAppRunning

Purpose

Return target application running status

Prototype

```
int xPCIsAppRunning(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

If the target application is stopped, the `xPCIsAppRunning` function returns `0`. If the target application is running, this function returns `1`. If there is an error, this function returns `0`.

Description

The `xPCIsAppRunning` function returns `1` or `0` depending on whether the target application is stopped or running. If there is an error, use the function `xPCGetLastError` to check for the error string constant.

See Also

API function `xPCIsOverloaded`

Target object property `Status`

xPCIsOverloaded

Purpose Return target PC overload status

Prototype

```
int xPCIsOverloaded(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return If the application is running properly, the `xPCIsOverloaded` function returns `1`. If the CPU is overloaded, the `xPCIsOverloaded` function returns `0`. In case of error, this function returns `-1`.

Description The `xPCIsOverloaded` function returns 1 if the target application is running properly and has not overloaded the CPU. It returns 0 if the target application has overloaded the target PC (CPU Overload).

See Also API function `xPCIsAppRunning`

Target object property `CPUoverload`

Purpose

Return data acquisition status for scope

Prototype

```
int xPCIsScFinished(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

If a scope finishes a data acquisition cycle, the `xPCIsScFinished` function returns `1`. If the scope is in the process of acquiring data, this function returns `0`. If there is an error, this function returns `-1`.

Description

The `xPCIsScFinished` function returns a Boolean value depending on whether scope *scNum* is finished (state of `SCST_FINISHED`) or not. You can also call this function for scopes of type target; however, because target scopes restart immediately, it is almost impossible to find these scopes in the finished state. Use the `xPCGetScope` function to get the scope number.

See Also

API function `xPCScGetState`

Scope object property `Status`

xPCLoadApp

Purpose

Load target application onto target PC

Prototype

```
void xPCLoadApp(int port, const char *pathstr,
const char *filename);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
pathstr	Enter the path to the target application file.
filename	Enter the name of a compiled target application (`*.dlm`) without the file extension.

Description

The `xPCLoadApp` function loads the compiled target application to the target PC. *pathstr* must not contain the trailing backslash. *pathstr* can be set to NULL or to the string `'nopath'` if the application is in the current directory. The variable *filename* must not contain the target application extension.

Before returning, `xPCLoadApp` waits for a certain amount of time before checking whether the model initialization is complete. In the case where the model initialization is incomplete, `xPCLoadApp` returns a time-out error to indicate a connection problem (for example, `ETCPREAD`). By default, `xPCLoadApp` checks for target readiness five times, with each attempt taking approximately 1 second (less if the target is ready). However, in the case of larger models or models requiring longer initialization (for example, those with thermocouple boards), the default of about 5 seconds might be insufficient and a spurious time-out can be generated. The functions `xPCGetLoadTimeOut` and `xPCSetLoadTimeOut` control the number of attempts made.

See Also

API functions `xPCStartApp`, `xPCStopApp`, `xPCUnloadApp`, `xPCSetLoadTimeOut`, `xPCGetLoadTimeOut`

Target object method `load`

Purpose Restore parameter values

Prototype

```
void xPCLoadParamSet(int port, const char *filename);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
filename	Enter the name of the file that contains the saved parameters.

Description The `xPCLoadParamSet` function restores the target application parameter values saved in the file *filename*. This file must be located on a local drive of the target PC. The parameter file must have been saved from a previous call to `xPCSaveParamSet`.

See Also API function `xPCSaveParamSet`

xPCMaxLogSamples

Purpose

Return maximum number of samples that can be in log buffer

Prototype

```
int xPCMaxLogSamples(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCMaxLogSamples` function returns the total number of samples. If there is an error, this function returns `-1`.

Description

The `xPCMaxLogSamples` function returns the total number of samples that can be returned in the logging buffers.

Note that the target application must be stopped before you get the number.

See Also

API functions `xPCNumLogSamples`, `xPCNumLogWraps`, `xPCGetStateLog`, `xPCGetOutputLog`, `xPCGetTETLog`, `xPCGetTimeLog`

Target object property `MaxLogSamples`

Purpose

Copy maximum task execution time to array

Prototype

```
void xPCMaximumTET(int port, double *data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
data	Array of at least two doubles.

Description

The `xPCMaximumTET` function gets the maximum task execution time (TET) that was achieved during the previous target application run. This function also returns the time at which the maximum TET was achieved. The `xPCMaximumTET` function then copies these values into the *data* array. The maximum TET value is copied into the first element, and the time at which it was achieved is copied into the second element.

See Also

API functions `xPCMinimumTET`, `xPCAverageTET`

Target object property `MaxTET`

xPCMinimumTET

Purpose

Copy minimum task execution time to array

Prototype

```
void xPCMinimumTET(int port, double *data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
data	Array of at least two doubles.

Description

The `xPCMinimumTET` function gets the minimum task execution time (TET) that was achieved during the previous target application run. This function also returns the time at which the minimum TET was achieved. The `xPCMinimumTET` function then copies these values into the *data* array. The minimum TET value is copied into the first element, and the time at which it was achieved is copied into the second element.

See Also

API functions `xPCMaximumTET`, `xPCAverageTET`

Target object property `MinTET`

Purpose

Return number of samples in log buffer

Prototype

```
int xPCNumLogSamples(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCNumLogSamples` function returns the number of samples in the log buffer. If there is an error, this function returns `-1`.

Description

The `xPCNumLogSamples` function returns the number of samples in the log buffer. In contrast to `xPCMaxLogSamples`, which returns the maximum number of samples that can be logged (because of buffer size constraints), xPCNumLogSamples returns the number of samples actually logged.

Note that the target application must be stopped before you get the number.

See Also

API functions `xPCGetStateLog`, `xPCGetOutputLog`, `xPCGetTETLog`, `xPCGetTimeLog`, `xPCMaxLogSamples`

xPCNumLogWraps

Purpose Return number of times log buffer wraps

Prototype

```
int xPCNumLogWraps(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return The `xPCNumLogWraps` function returns the number of times the log buffer wraps. If there is an error, this function returns `-1`.

Description The `xPCNumLogWraps` function returns the number of times the log buffer wraps.

Note that the target application must be stopped before you get the number.

See Also API functions `xPCNumLogSamples`, `xPCMaxLogSamples`, `xPCGetStateLog`, `xPCGetOutputLog`, `xPCGetTETLog`, `xPCGetTimeLog`

Target object property `NumLogWraps`

xPCOpenConnection

Purpose

Open connection to target PC

Prototype

```
void xPCOpenConnection(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Description

The `xPCOpenConnection` function opens a connection to the target PC whose data is indexed by *port*. Before calling this function, set up the target information by calling `xPCRegisterTarget`. A call to either `xPCOpenSerialPort` or `xPCOpenTcpIpPort` can also set up the target information. If the port is already open, calling this function has no effect.

See Also

API functions `xPCOpenTcpIpPort`, `xPCClosePort`, `xPCReOpenPort`, `xPCTargetPing`, `xPCCloseConnection`, `xPCRegisterTarget`

xPCOpenSerialPort

Purpose

Open RS-232 connection to xPC Target system

Prototype

```
int xPCOpenSerialPort(int comPort, int baudRate);
```

Arguments

comPort	Index of the COM port to be used (`0` is COM1, `1` is COM2, and so forth).
baudRate	*baudRate* must be one of the following values: 1200, 2400, 4800, 9600, 19200, 38400, 57600, or 115200.

Return

The `xPCOpenSerialPort` function returns the port value for the connection. If there is an error, this function returns `-1`.

Description

The `xPCOpenSerialPort` function initiates an RS-232 connection to an xPC Target system. It returns the port value for the connection. Be sure to pass this value to all the xPC Target API functions that require a port value.

If you enter a value of 0 for *baudRate*, this function sets the baud rate to the default value (115200).

See Also

API functions `xPCOpenTcpIpPort`, `xPCClosePort`, `xPCReOpenPort`, `xPCTargetPing`, `xPCOpenConnection`, `xPCCloseConnection`, `xPCRegisterTarget`, `xPCDeRegisterTarget`

Purpose

Open TCP/IP connection to xPC Target system

Prototype

```
int xPCOpenTcpIpPort(const char *ipAddress, const char
*ipPort);
```

Arguments

ipAddress	Enter the IP address of the target as a dotted decimal string. For example, "192.168.0.10".
ipPort	Enter the associated IP port as a string. For example, "22222".

Return

The `xPCOpenTcpIpPort` function returns a nonnegative integer that you can then use as the port value for any xPC Target API function that requires it. If this operation fails, this function returns `-1`.

Description

The `xPCOpenTcpIpPort` function opens a connection to the TCP/IP location specified by the IP address. It returns a nonnegative integer if it succeeds. Use this integer as the *ipPort* variable in the xPC Target API functions that require a port value. The global error number is also set, which you can get using `xPCGetLastError`.

See Also

API functions `xPCOpenSerialPort`, `xPCClosePort`, `xPCReOpenPort`, `xPCTargetPing`

xPCProtocol.Close

Purpose Close RS-232 or TCP/IP communication connection

Prototype

```
long Close();
```

Member Of `XPCAPICOMLib.xPCProtocol`

Return If there is an error, this method returns `0`. The `xPCProtocol.Close` method returns `-1` upon success.

Description The `xPCProtocol.Close` method closes the communication channel opened by `xPCProtocol.RS232Connect` or `xPCProtocol.TcpIpConnect`.

Purpose

Return current time-out value for target application initialization

Prototype

```
long GetLoadTimeOut();
```

Member Of

`XPCAPICOMLib.xPCProtocol`

Return

The `xPCProtocol.GetLoadTimeOut` method returns the number of seconds allowed for the initialization of the target application. If there is an error, this method returns `-1`.

Description

The `xPCProtocol.GetLoadTimeOut` method returns the number of seconds allowed for the initialization of the target application.

When you load a new target application onto the target PC, the method `xPCTarget.LoadApp` waits for a certain amount of time before checking to see whether the initialization of the target application is complete. In the case where initialization of the target application is not complete, the method `xPCTarget.LoadApp` returns a time-out error. By default, `xPCTarget.LoadApp` checks five times to see whether the target application is ready, with each attempt taking about 1 second. However, in the case of larger models or models requiring longer initialization (for example, those with thermocouple boards), the default of about 5 seconds might not be sufficient and a spurious time-out is generated. The method `xPCProtocol.SetLoadTimeOut` sets the time-out to a different number.

Use the `xPCProtocol.GetLoadTimeOut` method if you suspect that the current number of seconds (the time-out value) is too short. Then use the `xxPCProtocol.SetLoadTimeOut` method to set the time-out to a higher number.

xPCProtocol.GetxPCErrorMsg

Purpose

Return error string

Prototype

```
BSTR GetxPCErrorMsg();
```

Member Of

`XPCAPICOMLib.xPCProtocol`

Return

The `xPCProtocol.GetxPCErrorMsg` method returns the string for the last reported error.

Description

The `xPCProtocol.GetxPCErrorMsg` method returns the string of the last error reported by another COM API method. This value is reset every time you call a new method. Therefore, you should check this constant value immediately after a call to an API COM method. You can use this method in conjunction with the `xPCProtocol.isxPCError` method, which detects that an error has occurred.

See Also

API function `xPCProtocol.isxPCError`

Purpose

Initialize xPC Target API DLL

Prototype

```
long Init();
```

Member Of

`XPCAPICOMLib.xPCProtocol`

Return

If there is an error, this method returns `-1`. The `xPCProtocol.Init` method returns `0` upon success.

Description

The `xPCProtocol.Init` method initializes the xPC Target API by loading the xPC Target DLL, `xpcapi.dll`, into memory. To load `xpcapi.dll` into memory, the method requires that the `xpcapi.dll` file be in one of the following directories:

- The directory in which the application is loaded
- The current directory
- The Windows system directory

xPCProtocol.isxPCError

Purpose Return error status

Prototype `long isxPCError();`

Member Of `XPCAPICOMLIB.xPCProtocol`

Return The `xPCProtocol.isxPCError` method returns the error status. If there is an error, this method returns `1`. The `xPCProtocol.isxPCError` method returns `0` upon success.

Description The `xPCProtocol.isxPCError` method returns the error status. Use this method to check for any errors that might occur after a call to any of the `xPCProtocol` class methods. If there is an error, call the `xPCProtocol.GetxPCErrorMsg` to get the string for the error.

See Also API function `xPCProtocol.GetxPCErrorMsg`

Purpose Contain communication channel index

Prototype `long Port();`

Member Of `XPCAPICOMLIB.xPCProtocol`

Return The `xPCProtocol.Port` property returns a positive number (the communication channel index) if the connection succeeds. The method returns a nonpositive number if the connection does not succeed.

Description The `xPCProtocol.Port` property contains the communication channel index if connection with the target PC succeeds. Note that you only need to use this property when working with a model-specific COM library that you generate from a Simulink model. See "Model-Specific COM Interface Library (model_nameCOMiface.dll)" on page 3-18.

xPCProtocol.Reboot

Purpose Reboot target PC

Prototype `long Reboot();`

Member Of `XPCAPICOMLib.xPCProtocol`

Return If there is an error, this method returns `0`. The `xPCProtocol.Reboot` method returns `-1` upon success.

Description The `xPCProtocol.Reboot` method reboots the target PC. This function does not close the connection to the target PC. You should explicitly close the connection, then reestablish the connection once the target PC has rebooted. Use the methods `xPCProtocol.RS232Connect` or `xPCProtocol.TcpIpConnect` to reestablish the connection.

Purpose

Open RS-232 connection to target PC

Prototype

```
long RS232Connect(long comport, long baudrate);
```

Member Of

`XPCAPICOMLib.xPCProtocol`

Arguments

[in] *comport*	Index of the COM port to be used (`0` is COM1, `1` is COM2, and so forth).
[in] *baudrate*	*baudrate* must be one of the following values: 1200, 2400, 4800, 9600, 19200, 38400, 57600, or 115200.

Return

The `xPCProtocol.RS232Connect` method returns the port value for the connection. If the connection succeeds, this method returns `-1`. If the connection fails, the `xPCProtocol.RS232Connect` method returns `0`.

Description

The `xPCProtocol.RS232Connect` method initiates an RS-232 connection to an xPC Target system. It returns the port value for the connection. Be sure to pass this value to all the xPC Target API functions that require a port value.

If you enter a value of 0 for *baudrate*, this function sets the baud rate to the default value (115200).

xPCProtocol.SetLoadTimeOut

Purpose Change initialization time-out value

Prototype `long SetLoadTimeOut(long timeOut);`

Member Of `XPCAPICOMLib.xPCProtocol`

Arguments

[in] *timeOut*	Enter the new initialization time-out value.

Return If there is an error, this method returns `0`. The `xPCProtocol.SetLoadTimeOut` method returns `-1` upon success. To get the string description for the error, use `xPCProtocol.GetxPCErrorMsg`.

Description The `xPCProtocol.SetLoadTimeOut` method changes the time-out value for initialization. The *timeOut* value is the time the method `xPCTarget.LoadApp` waits to check whether the model initialization for a new application is complete before returning. It enables you to set the number of initialization attempts to be made before signaling a time-out. When a new target application is loaded onto the target PC, the method `xPCTarget.LoadApp` waits for a certain time to check whether the model initialization is complete before returning. If the model initialization is incomplete within the allotted time, `xPCTarget.LoadApp` returns a time-out error.

By default, `xPCTarget.LoadApp` checks for target readiness five times, with each attempt taking approximately 1 second (less if the target is ready). However, in the case of larger models or models requiring longer initialization (for example, models with thermocouple boards), the default of about 5 seconds might be insufficient and a spurious time-out can be generated.

Purpose

Ping target PC

Prototype

```
long TargetPing;
```

Member Of

`XPCAPICOMLIB.xPCProtocol`

Return

The `xPCProtocol.TargetPing` method returns `1` if it successfully reaches the target. If there is an error, the method returns `0`.

Description

The `xPCProtocol.TargetPing` method pings the target PC and returns `1` or `0` depending on whether the target responds or not. All errors, such as the inability to connect to the target, are ignored.

If you are using TCP/IP, note that `xPCProtocol.xPCTargetPing` will cause the target PC to close the TCP/IP connection. You can use `xPCProtocol.TcpIpConnect` to reconnect. You can also use this `xPCProtocol.xPCTargetPing` feature to close the target PC connection in the event of an aborted TCP/IP connection (for example, if your host side program crashes).

xPCProtocol.TcpIpConnect

Purpose Open TCP/IP connection to target PC

Prototype

```
long TcpIpConnect(BSTR TargetIpAddress, BSTR TargetPort);
```

Member Of XPCAPICOMLIB.xPCProtocol

Arguments

[in] *TargetIpAddress*	Enter the IP address of the target as a dotted decimal string. For example, "192.168.0.10".
[in] *TargetPort*	Enter the associated IP port as a string. For example, "22222".

Return If the connection succeeds, this method returns -1. If the connection fails, the `xPCProtocol.TcpIpConnect` method returns 0.

Description The `xPCProtocol.TcpIpConnect` method opens a connection to the TCP/IP location specified by the IP address. Use this integer as the *TargetPort* variable in the xPC Target COM API functions that require a port value.

Purpose Unload xPC Target API DLL from memory

Prototype

```
long Term();
```

Member Of `XPCAPICOMLib.xPCProtocol`

Return This method always returns `-1`.

Description The `xPCProtocol.Term` method unloads the xPC Target API DLL (`xpcapi.dll`) from memory. You must call this method when you want to terminate your COM API application.

xPCReboot

Purpose

Reboot target PC

Prototype

```
void xPCReboot(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Description

The `xPCReboot` function reboots the target PC. This function returns nothing. This function does not close the connection to the target PC. You should either explicitly close the port or call `xPCReOpenPort` once the target PC has rebooted.

See Also

API function `xPCReOpenPort`

Target object method `reboot`

Purpose

Reopen communication channel

Prototype

```
int xPCReOpenPort(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return

The `xPCReOpenPort` function returns 0 if it successfully reopens a connection. If there is an error, this function returns `-1`.

Description

The `xPCReOpenPort` function reopens the communications channel pointed to by *port*. The difference between this function and `xPCOpenSerialPort` or `xPCOpenTcpIpPort` is that `xPCReOpenPort` uses the already existing settings, while the other functions need to be set up properly.

See Also

API functions `xPCOpenTcpIpPort`, `xPCClosePort`

xPCRegisterTarget

Purpose

Register target with xPC Target API library

Prototype

```
int xPCRegisterTarget(int commType, const char *ipAddress,
const char *ipPort, int comPort, int baudRate);
```

Arguments

commType	Specify the communication type (TCP/IP or RS-232) between the host and the target.
ipAddress	Enter the IP address of the target as a dotted decimal string. For example, "192.168.0.10".
ipPort	Enter the associated IP port as a string. For example, "22222".
comPort	*comPort* and *baudRate* are as in xPCOpenSerialPort.
baudRate	The *baudRate* must be one of the following values: 1200, 2400, 4800, 9600, 19200, 38400, 57600, or 115200.

Return

The xPCRegisterTarget function returns the port number.

Description

The xPCRegisterTarget function works similarly to xPCOpenSerialPort and xPCOpenTcpIpPort, except that it does not try to open a connection to the target PC. In other words, xPCOpenSerialPort or xPCOpenTcpIpPort is equivalent to calling xPCRegisterTarget with the appropriate parameters, followed by a call to xPCOpenConnection.

Use the constants COMMTYP_TCPIP and COMMTYP_RS232 for *commType*. If *commType* is set to COMMTYP_RS232, the function ignores *ipAddress* and *ipPort*. Analogously, the function ignores *comPort* and *baudRate* if *commType* is set to COMMTYP_TCPIP.

If you enter a value of 0 for *baudRate*, this function sets the baud rate to the default value (115200).

See Also

API functions `xPCDeRegisterTarget`, `xPCOpenTcpIpPort`, `xPCOpenSerialPort`, `xPCClosePort`, `xPCReOpenPort`, `xPCOpenConnection`, `xPCCloseConnection`, `xPCTargetPing`

xPCRemScope

Purpose

Remove scope

Prototype

```
void xPCRemScope(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Description

The `xPCRemScope` function removes the scope with number *scNum*. Attempting to remove a nonexistent scope causes an error. For a list of existing scopes, see `xPCGetScopes`. Use the `xPCGetScope` function to get the scope number.

See Also

API functions `xPCAddScope`, `xPCScRemSignal`, `xPCGetScopes`

Target object method `remscope`

xPCSaveParamSet

Purpose Save parameter values of target application

Prototype

```
void xPCSaveParamSet(int port, const char *filename);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
filename	Enter the name of the file to contain the saved parameters.

Description

The `xPCSaveParamSet` function saves the target application parameter values in the file *filename*. This function saves the file on a local drive of the current target PC. You can later reload these parameters with the `xPCLoadParamSet` function.

You might want to save target application parameter values if you change these parameter values while the application is running in real time. Saving these values enable you to easily recreate target application parameter values from a number of application runs.

See Also API function `xPCLoadParamSet`

xPCScAddSignal

Purpose

Add signal to scope

Prototype

```
void xPCScAddSignal(int port, int scNum, int sigNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
sigNum	Enter a signal number.

Description

The `xPCScAddSignal` function adds the signal with number *sigNum* to the scope *scNum*. The signal should not already exist in the scope. You can use `xPCScGetSignals` to get a list of the signals already present. Use the function `xPCGetScope` to get the scope number. Use the `xPCGetSignalIdx` function to get the signal number.

See Also

API functions `xPCScRemSignal`, `xPCAddScope`, `xPCRemScope`, `xPCGetScopes`

Scope object method `addsignal`

Purpose

Copy scope data to array

Prototype

```
void xPCScGetData(int port, int scNum, int
signal_id, int start,
int numsamples, int decimation, double *data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
signal_id	Enter a signal number.
start	Enter the first sample from which data retrieval is to start
numsamples	Enter the number of samples retrieved with a decimation of *decimation*, starting from the *start* value.
decimation	Enter a value such that every *decimation* sample is retrieved in a scope window.
data	The data is available in the array *data*, starting from sample *start*.

Description

The `xPCScGetData` function gets the data used in a scope. Use this function for scopes of type `SCTYPE_HOST`. The scope must be either in state `"Finished"` or in state `"Interrupted"` for the data to be retrievable. (Use the `xPCScGetState` function to check the state of the scope.) The data must be retrieved one signal at a time. The calling function must allocate the space ahead of time to store the scope data. *data* must be an array of doubles, regardless of the data type of the signal to be retrieved. Use the function `xPCScGetSignals` to get the list of signals in the scope for *signal_id*. Use the function `xPCGetScope` to get the scope number for *scNum*.

See Also

API functions `xPCGetScope`, `xPCScGetState`, `xPCScGetSignals`

Scope object property `Data`

xPCScGetDecimation

Purpose

Return decimation of scope

Prototype

```
int xPCScGetDecimation(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

The `xPCScGetDecimation` function returns the decimation of scope *scNum*. If there is an error, this function returns `-1`.

Description

The `xPCScGetDecimation` function gets the decimation of scope *scNum*. The decimation is a number, `N`, meaning every `N`th sample is acquired in a scope window. Use the `xPCGetScope` function to get the scope number.

See Also

API function `xPCScSetDecimation`

Scope object property `Decimation`

xPCScGetNumPrePostSamples

Purpose

Get number of pre- or posttriggering samples before triggering scope

Prototype

```
int xPCScGetNumPrePostSamples(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

The `xPCScGetNumPrePostSamples` function returns the number of samples for pre- or posttriggering for scope *scNum*. If an error occurs, this function returns the minimum integer value (`-2147483647-1`).

Description

The `xPCScGetNumPrePostSamples` function gets the number of samples for pre- or posttriggering for scope *scNum*. A negative number implies pretriggering, whereas a positive number implies posttriggering samples. Use the `xPCGetScope` function to get the scope number.

See Also

API function `xPCScSetNumPrePostSamples`

Scope object property `NumPrePostSamples`

xPCScGetNumSamples

Purpose Get number of samples in one data acquisition cycle

Prototype

```
int xPCScGetNumSamples(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return The `xPCScGetNumSamples` function returns the number of samples in the scope *scNum*. If there is an error, this function returns `-1`.

Description The `xPCScGetNumSamples` function gets the number of samples in one data acquisition cycle for scope *scNum*. Use the `xPCGetScope` function to get the scope number.

See Also API function `xPCScSetNumSamples`

Scope object property `NumSamples`

xPCScGetSignals

Purpose

Copy list of signals to array

Prototype

```
void xPCScGetSignals(int port, int scNum, int *data);
```

Arguments

port	Value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
data	Integer array allocated by the caller as a list containing the signal identifiers, terminated by `-1`.

Description

The `xPCScGetSignals` function gets the list of signals defined for scope *scNum*. You can use the constant `MAX_SIGNALS`, defined in `xpcapiconst.h`, as the size of *data*. Use the `xPCGetScope` function to get the scope number.

See Also

API functions `xPCScGetData`, `xPCGetScopes`

Scope object property `Signals`

Purpose

Get start time for last data acquisition cycle

Prototype

```
double xPCScGetStartTime(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

The `xPCScGetStartTime` function returns the start time for the last data acquisition cycle of a scope. If there is an error, this function returns `-1`.

Description

The `xPCScGetStartTime` function gets the time at which the last data acquisition cycle for scope *scNum* started. This is only valid for scopes of type `SCTYPE_HOST`. Use the `xPCGetScope` function to get the scope number.

See Also

API functions `xPCScGetNumSamples`, `xPCScGetDecimation`

Scope object property `StartTime`

xPCScGetState

Purpose

Get state of scope

Prototype

```
int xPCScGetState(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

The `xPCScGetState` function returns the state of scope *scNum*. If there is an error, this function returns `-1`.

Description

The `xPCScGetState` function gets the state of scope *scNum*, or `-1` upon error. Use the `xPCGetScope` function to get the scope number.

Constants to find the scope state, defined in `xpcapiconst.h`, have the following meanings:

Constant	Value	Description
SCST_WAITTOSTART	0	Scope is ready and waiting to start.
SCST_PREACQUIRING	5	Scope acquires a predefined number of samples before triggering.
SCST_WAITFORTRIG	1	After a scope is finished with the preacquiring state, it waits for a trigger. If the scope does not preacquire data, it enters the wait for trigger state.
SCST_ACQUIRING	2	Scope is acquiring data. The scope enters this state when it leaves the wait for trigger state.

Constant	Value	Description
SCST_FINISHED	3	Scope is finished acquiring data when it has attained the predefined limit.
SCST_INTERRUPTED	4	The user has stopped (interrupted) the scope.

See Also

API functions xPCScStart, xPCScStop

Scope object property Status

xPCScGetTriggerLevel

Purpose Get trigger level for scope

Prototype

```
double xPCScGetTriggerLevel(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.
scNum	Enter the scope number.

Return The xPCScGetTriggerLevel function returns the scope trigger level.

Description The xPCScGetTriggerLevel function gets the trigger level for scope *scNum*. Use the xPCGetScope function to get the scope number.

See Also API functions xPCScSetTriggerLevel, xPCScSetTriggerSlope, xPCScGetTriggerSlope, xPCScSetTriggerSignal, xPCScGetTriggerSignal, xPCScSetTriggerScope, xPCScGetTriggerScope, xPCScSetTriggerMode, xPCScGetTriggerMode

Scope object property TriggerLevel

Purpose

Get trigger mode for scope

Prototype

```
int xPCScGetTriggerMode(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

The `xPCScGetTriggerMode` function returns the scope trigger mode. If there is an error, this function returns `-1`.

Description

The `xPCScGetTriggerMode` function gets the trigger mode for scope *scNum*. Use the `xPCGetScope` function to get the scope number. Use the constants defined in `xpcapiconst.h` to interpret the trigger mode. These constants include the following:

Constant	Value	Description
TRIGMD_FREERUN	0	There is no trigger mode. The scope always triggers when it is ready to trigger, regardless of the circumstances.
TRIGMD_SOFTWARE	1	Only a user can trigger the scope. It is always possible for a user to trigger the scope; however, if you set the scope to this trigger mode, user intervention is the only way to trigger the scope. No other triggering is possible.

Constant	Value	Description
TRIGMD_SIGNAL	2	Signal must cross a value before the scope is triggered.
TRIGMD_SCOPE	3	Scope is triggered by another scope at the trigger point of the triggering scope, modified by the value of triggerscopesample (see scopedata).

See Also

API functions xPCScSetTriggerLevel, xPCScGetTriggerLevel, xPCScSetTriggerSlope, xPCScGetTriggerSlope, xPCScSetTriggerSignal, xPCScGetTriggerSignal, xPCScSetTriggerScope, xPCScGetTriggerScope, xPCScSetTriggerMode

Scope object method trigger

Scope object property TriggerMode

Purpose

Get trigger scope

Prototype

```
int xPCScGetTriggerScope(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

The `xPCScGetTriggerScope` function returns a trigger scope. If there is an error, this function returns `-1`.

Description

The `xPCScGetTriggerScope` function gets the trigger scope for scope *scNum*. Use the `xPCGetScope` function to get the scope number.

See Also

API functions `xPCScSetTriggerLevel`, `xPCScGetTriggerLevel`, `xPCScSetTriggerSlope`, `xPCScGetTriggerSlope`, `xPCScSetTriggerSignal`, `xPCScGetTriggerSignal`, `xPCScSetTriggerMode`, `xPCScGetTriggerMode`

Scope object property `TriggerScope`

xPCScGetTriggerScopeSample

Purpose

Get sample number for triggering scope

Prototype

```
int xPCScGetTriggerScopeSample(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

The `xPCScGetTriggerScopeSample` function returns a nonnegative integer for a real sample, and `-1` for the special case where triggering is at the end of the data acquisition cycle for a triggering scope. If there is an error, this function returns `INT_MIN` (`-2147483647-1`).

Description

The `xPCScGetTriggerScopeSample` function gets the number of samples a triggering scope (*scNum*) acquires before starting data acquisition on a second scope. This value is a nonnegative integer for a real sample, and `-1` for the special case where triggering is at the end of the data acquisition cycle for a triggering scope. Use the `xPCGetScope` function to get the scope number for the trigger scope.

See Also

API functions `xPCScSetTriggerLevel`, `xPCScGetTriggerLevel`, `xPCScSetTriggerSlope`, `xPCScGetTriggerSlope`, `xPCScSetTriggerSignal`, `xPCScGetTriggerSignal`, `xPCScSetTriggerScope`, `xPCScGetTriggerScope`, `xPCScSetTriggerMode`, `xPCScGetTriggerMode`, `xPCScSetTriggerScopeSample`

Scope object property `TriggerSample`

Purpose Get trigger signal for scope

Prototype

```
int xPCScGetTriggerSignal(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return The `xPCScGetTriggerSignal` function returns the scope trigger signal. If there is an error, this function returns `-1`.

Description The `xPCScGetTriggerSignal` function gets the trigger signal for scope *scNum*. Use the `xPCGetScope` function to get the scope number for the trigger scope.

See Also API functions `xPCScSetTriggerLevel`, `xPCScGetTriggerLevel`, `xPCScSetTriggerSlope`, `xPCScGetTriggerSlope`, `xPCScSetTriggerSignal`, `xPCScSetTriggerScope`, `xPCScGetTriggerScope`, `xPCScSetTriggerMode`, `xPCScGetTriggerMode`

Scope object method `trigger`

Scope object property `TriggerSignal`

xPCScGetTriggerSlope

Purpose

Get trigger slope for scope

Prototype

```
int xPCScGetTriggerSlope(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

The `xPCScGetTriggerSlope` function returns the scope trigger slope. If there is an error, this function returns `-1`.

Description

The `xPCScGetTriggerSlope` function gets the trigger slope of scope *scNum*. Use the `xPCGetScope` function to get the scope number for the trigger scope. Use the constants defined in `xpcapiconst.h` to interpret the trigger slope. These constants have the following meanings:

Constant	Value	Description
`TRIGSLOPE_EITHER`	0	The trigger slope can be either rising or falling.
`TRIGSLOPE_RISING`	1	The trigger slope must be rising when the signal crosses the trigger value.
`TRIGSLOPE_FALLING`	2	The trigger slope must be falling when the signal crosses the trigger value.

See Also

API functions `xPCScSetTriggerLevel`, `xPCScGetTriggerLevel`, `xPCScSetTriggerSlope`, `xPCScSetTriggerSignal`, `xPCScGetTriggerSignal`, `xPCScSetTriggerScope`, `xPCScGetTriggerScope`, `xPCScSetTriggerMode`, `xPCScGetTriggerMode`

Scope object method `trigger`

Scope object properties `TriggerMode`, `TriggerSlope`

xPCScGetType

Purpose

Get type of scope

Prototype

```
int xPCScGetType(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

The `xPCScGetType` function returns the scope type. If there is an error, this function returns `-1`.

Description

The `xPCScGetType` function gets the type (`SCTYPE_HOST` for host, `SCTYPE_TARGET` for target, or `SCTYPE_FILE` for file) of scope *scNum*. Use the constants defined in `xpcapiconst.h` to interpret the return value. A scope of type `SCTYPE_HOST` is displayed on the host PC while a scope of type `SCTYPE_TARGET` is displayed on the target PC screen. A scope of type `SCTYPE_FILE` is stored on a storage medium. Use the `xPCGetScope` function to get the scope number.

See Also

API functions `xPCAddScope`, `xPCRemScope`

Scope object property `Type`

Purpose Create new scope of type file

Prototype `long AddFileScope(long scNum);`

Member Of `XPCAPICOMLib.xPCScopes`

Arguments

[in] *scNum*	Enter a number for a new scope. Values are `1`, `2`, `3. . .`

Return If there is an error, this method returns `0`. The `xPCScopes.AddFileScope` method returns `-1` upon success.

Description The `xPCScopes.AddFileScope` method creates a new scope of type file on the target PC.

Calling the `xPCScopes.AddFileScope` method with *scNum* having the number of an existing scope produces an error. Use `xPCScopes.GetScopes` to find the numbers of existing scopes.

xPCScopes.AddHostScope

Purpose

Create new scope of type host

Prototype

```
long AddHostScope(long scNum);
```

Member Of

`XPCAPICOMLib.xPCScopes`

Arguments

[in] *scNum*	Enter a number for a new scope. Values are `1, 2, 3. . .`

Return

If there is an error, this method returns `0`. The `xPCScopes.AddHostScope` method returns `-1` upon success.

Description

The `xPCScopes.AddHostScope` method creates a new scope of type host on the target PC.

Calling the `xPCScopes.AddHostScope` method with *scNum* having the number of an existing scope produces an error. Use `xPCScopes.GetScopes` to find the numbers of existing scopes.

Purpose Create new scope of type target

Prototype `long AddTargetScope(long scNum);`

Member Of `XPCAPICOMLib.xPCScopes`

Arguments

[in] *scNum*	Enter a number for a new scope. Values are 1, 2, 3. . .

Return If there is an error, this method returns 0. The `xPCScopes.AddTargetScope` method returns -1 upon success.

Description If there is an error, this function returns 0. The `xPCScopes.AddTargetScope` method creates a new scope on the target PC.

Calling the `xPCScopes.AddTargetScope` method with *scNum* having the number of an existing scope produces an error. Use `xPCScopes.GetScopes` to find the numbers of existing scopes.

xPCScopes.GetScopes

Purpose Get and copy list of scope numbers

Prototype

```
VARIANT GetScopes(long size);
```

Member Of XPCAPICOMLib.xPCScopes

Arguments

[in] *size*	Specify the size of the VARIANT array returned. This argument must be greater than MAX_SCOPES-1. The elements in the array consist of a list of unsorted integers, terminated by -1.

Return The xPCScopes.GetScopes method returns a VARIANT array with elements containing a list of scope numbers from the target application.

Description The xPCScopes.GetScopes method gets a VARIANT array with elements containing a list of scope numbers currently defined for the target application. Specify the size of the VARIANT array returned. This size must be greater than the maximum number of scopes -1, up to a maximum of 30 scopes. The elements in the array consist of a list of unsorted integers, terminated by -1.

Purpose Get error string

Prototype

```
BSTR GetxPCError();
```

Member Of `XPCAPICOMLib.xPCScopes`

Return The `xPCScopes.GetxPCError` method returns the string for the last reported error. If there is no error, this function returns 0.

Description The `xPCScopes.GetxPCError` method gets the string of the last reported error by another COM API method. This value is reset every time you call a new method. Therefore, you should check this constant value immediately after a call to an API COM method. You can use this method in conjunction with the `xPCScopes.isxPCError` method, which detects that an error has occurred.

See Also API function `xPCScopes.isxPCError`

xPCScopes.Init

Purpose

Initialize scope object to communicate with target PC

Prototype

```
long Init(IxPCProtocol* xPCProtocol);
```

Member Of

`XPCAPICOMLib.xPCScopes`

Arguments

[in] `xPCProtocol`	Specify the communication port of the target PC object for which the scope is to be initialized.

Return

If there is an error, this method returns `-1`. Otherwise, the `xPCScopes.Init` method returns `-1`.

Description

The `xPCScopes.Init` method initializes the scope object to communicate with the target PC referenced by the `xPCProtocol` object.

Purpose

Get data acquisition status for scope

Prototype

`long IsScopeFinished(long *scNum*);`

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return

If a scope finishes a data acquisition cycle, the `xPCScopes.IsScopeFinished` method returns `1`. If the scope is in the process of acquiring data, this method returns `0`. If there is an error, this method returns `-1`.

Description

The `xPCScopeos.IsScopeFinished` method gets a `1` or `0` depending on whether scope *scNum* is finished (state of `SCST_FINISHED`) or not. You can also call this function for scopes of type target; however, because target scopes restart immediately, it is almost impossible to find these scopes in the finished state.

xPCScopes.isxPCError

Purpose Get error status

Prototype

```
long isxPCError();
```

Member Of `XPCAPICOMLIB.xPCScopes`

Return The `xPCScopes.isxPCError` method returns the error status. If there is an error, this method returns `1`. The `xPCScopes.isxPCError` method returns `0` upon success.

Description The `xPCProtocol.isxPCError` method gets the error status. Use this method to check for any errors that might occur after a call to any of the `xPCScopes` class methods. If there is an error, call the `xPCScopes.GetxPCError` method to get the string for the error.

See Also API function `xPCScopes.GetxPCError`

Purpose Remove scope

Prototype `long RemScope(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return If there is an error, this method returns `0`. The `xPCScopes.RemScope` method returns `-1` upon success.

Description The `xPCScopes.RemScope` method removes the scope with number *scNum*. Attempting to remove a nonexistent scope causes an error. For a list of existing scopes, use `xPCScopes.GetScopes`.

xPCScopes.ScopeAddSignal

Purpose Add signal to scope

Prototype

```
long ScopeAddSignal(long scNum, long sigNum);
```

Member Of `XPCAPICOMLib.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
[in] *sigNum*	Enter a signal number.

Return If there is an error, this method returns `0`. The `xPCScopes.ScopeAddSignal` method returns `-1` upon success.

Description The `xPCScopes.ScopeAddSignal` method adds the signal with number *sigNum* to the scope *scNum*. The signal should not already exist in the scope. You can use `xPCScopes.ScopeGetSignals` to get a list of the signals already present. Use the `xPCTarget.GetSignalIdx` method to get the signal number.

Purpose

Copy scope data to array

Prototype

```
VARIANT ScopeGetData(long scNum, long signal_id,
long start,
long numsamples, long decimation);
```

Member Of

XPCAPICOMLIB.xPCScopes

Arguments

[in] *scNum*	Enter the scope number.
[in] *signal_id*	Enter a signal number.
[in] *start*	Enter the first sample from which data retrieval is to start.
[in] *numsamples*	Enter the number of samples retrieved with a decimation of *decimation*, starting from the *start* value.
[in] *decimation*	Enter a value such that every *decimation* sample is retrieved in a scope window.

Return

The xPCScopes.ScopeGetData method returns a VARIANT array with elements containing the data used in a scope.

Description

The xPCScopes.ScopeGetData method gets the data used in a scope. Use this function for scopes of type SCTYPE_HOST. The scope must be either in state Finished or in state Interrupted for the data to be retrievable. (Use the xPCScopes.ScopeGetState method to check the state of the scope.) The data must be retrieved one signal at a time. The calling function determines and allocates the space ahead of time to store the scope data. Use the method xPCScopes.ScopeGetSignals to get the list of signals in the scope for *signal_id*.

xPCScopes.ScopeGetDecimation

Purpose Get decimation of scope

Prototype `long ScopeGetDecimation(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return The `xPCScopes.ScopeGetDecimation` method returns the decimation of scope *scNum*. If there is an error, this function returns `-1`.

Description The `xPCScopes.ScopeGetDecimation` method gets the decimation of scope *scNum*. The decimation is a number, `N`, meaning every `N`th sample is acquired in a scope window.

Purpose

Get number of pre- or posttriggering samples before triggering scope

Prototype

`long ScopeGetNumPrePostSamples(long scNum);`

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return

The `xPCScopes.ScopeGetNumPrePostSamples` method returns the number of samples for pre- or posttriggering for scope *scNum*. If an error occurs, this method returns `-1`.

Description

The `xPCScopes.ScopeGetNumPrePostSamples` method gets the number of samples for pre- or posttriggering for scope *scNum*. A negative number implies pretriggering, whereas a positive number implies posttriggering samples.

xPCScopes.ScopeGetNumSamples

Purpose Get number of samples in one data acquisition cycle

Prototype

```
long ScopeGetNumSamples(long scNum);
```

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return The `xPCScopes.ScopeGetNumSamples` method returns the number of samples in the scope *scNum*.

Description The `xPCScopes.ScopeGetNumSamples` method gets the number of samples in one data acquisition cycle for scope *scNum*.

Purpose Get list of signals

Prototype `VARIANT ScopeGetSignals(long scNum, long size);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
[in] *size*	Enter an integer to allocate the number of elements to be returned in the `VARIANT` array. This size is required for the method to copy the list of signals into the `VARIANT` array. The maximum number of signals is 10.

Return The `xPCScopes.ScopeGetSignals` method returns a `VARIANT` array with elements consisting of the list of signals defined for a scope.

Description The `xPCScopes.ScopeGetSignals` method gets the list of signals defined for scope *scNum*. You can use the constant `MAX_SIGNALS`.

xPCScopes.ScopeGetStartTime

Purpose

Get last data acquisition cycle start time

Prototype

`double ScopeGetStartTime(long scNum);`

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return

The `xPCScopes.ScopeGetStartTime` method returns the start time for the last data acquisition cycle of a scope. If there is an error, this function returns `-1`.

Description

The `xPCScopes.ScopeGetStartTime` method gets the time at which the last data acquisition cycle for scope *scNum* started. This is only valid for scopes of type `SCTYPE_HOST`.

Purpose Get state of scope

Prototype `BSTR ScopeGetState(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return The `xPCScopes.ScopeGetState` method returns the state of scope *scNum*. If there is an error, this function returns `-1`.

Description The `xPCScopes.ScopeGetState` method gets the state of scope *scNum*, or `-1` upon error.

Constants to find the scope state have the following meanings:

Constant	Value	Description
SCST_WAITTOSTART	0	Scope is ready and waiting to start.
SCST_PREACQUIRING	5	Scope acquires a predefined number of samples before triggering.
SCST_WAITFORTRIG	1	After a scope is finished with the preacquiring state, it waits for a trigger. If the scope does not preacquire data, it enters the wait for trigger state.
SCST_ACQUIRING	2	Scope is acquiring data. The scope enters this state when it leaves the wait for trigger state.

Constant	Value	Description
SCST_FINISHED	3	Scope is finished acquiring data when it has attained the predefined limit.
SCST_INTERRUPTED	4	The user has stopped (interrupted) the scope.

Purpose

Get trigger level for scope

Prototype

```
double ScopeGetTriggerLevel(long scNum);
```

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return

The `xPCScopes.ScopeGetTriggerLevel` method returns the scope trigger level.

Description

The `xPCScopes.ScopeGetTriggerLevel` method gets the trigger level for scope *scNum*.

xPCScopes.ScopeGetTriggerMode

Purpose

Get trigger mode for scope

Prototype

`long ScopeGetTriggerMode(long scNum);`

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return

The `xPCScopes.ScopeGetTriggerMode` method returns the scope trigger mode. If there is an error, this method returns `-1`.

Description

The `xPCScopes.ScopeGetTriggerMode` method gets the trigger mode for scope *scNum*. Use the constants here to interpret the trigger mode:

Constant	Value	Description
TRIGMD_FREERUN	0	There is no trigger mode. The scope always triggers when it is ready to trigger, regardless of the circumstances.
TRIGMD_SOFTWARE	1	Only a user can trigger the scope. It is always possible for a user to trigger the scope; however, if you set the scope to this trigger mode, user intervention is the only way to trigger the scope. No other triggering is possible.

Constant	Value	Description
TRIGMD_SIGNAL	2	Signal must cross a value before the scope is triggered.
TRIGMD_SCOPE	3	Scope is triggered by another scope at the trigger point of the triggering scope, modified by the value of triggerscopesample (see scopedata).

See Also

API function xPCScopes.ScopeGetTriggerModeStr

xPCScopes.ScopeGetTriggerModeStr

Purpose

Get trigger mode as string

Prototype

`BSTR ScopeGetTriggerModeStr(long scNum);`

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return

The `xPCScopes.ScopeGetTriggerModeStr` method returns a string containing the trigger mode string.

Description

The `xPCScopes.ScopeGetTriggerModeStr` method gets the trigger mode string for scope *scNum*. This method returns one of the following strings.

Constant	Description
`FreeRun`	There is no trigger mode. The scope always triggers when it is ready to trigger, regardless of the circumstances.
`Software`	Only a user can trigger the scope. It is always possible for a user to trigger the scope; however, if you set the scope to this trigger mode, user intervention is the only way to trigger the scope. No other triggering is possible.
`Signal`	Signal must cross a value before the scope is triggered.
`Scope`	Scope is triggered by another scope at the trigger point of the triggering scope, modified by the value of `triggerscopesample` (see `scopedata`).

See Also

API function `xPCScopes.ScopeGetTriggerMode`

Purpose Get sample number for triggering scope

Prototype `long ScopeGetTriggerSample(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return The `xPCScopes.ScopeGetTriggerSample` method returns a nonnegative integer for a real sample, and `-1` for the special case where triggering is at the end of the data acquisition cycle for a triggering scope. If there is an error, this function returns `-1`.

Description The `xPCScopes.ScopeGetTriggerSample` method gets the number of samples a triggering scope (*scNum*) acquires before starting data acquisition on a second scope. This value is a nonnegative integer for a real sample, and `-1` for the special case where triggering is at the end of the data acquisition cycle for a triggering scope.

xPCScopes.ScopeGetTriggerSignal

Purpose Get trigger signal for scope

Prototype `long ScopeGetTriggerSignal(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return The `xPCScopes.ScopeGetTriggerSignal` method returns the scope trigger signal. If there is an error, this function returns `-1`.

Description The `xPCScopes.ScopeGetTriggerSignal` method gets the trigger signal for scope *scNum*.

Purpose

Get trigger slope for scope

Prototype

`long ScopeGetTriggerSlope(long scNum);`

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return

The `xPCScopes.ScopeGetTriggerSlope` method returns the scope trigger slope. If there is an error, this function returns `-1`.

Description

The `xPCScopes.ScopeGetTriggerSlope` method gets the trigger slope of scope *scNum*. Use the constants here to interpret the trigger slope:

String	Value	Description
TRIGSLOPE_EITHER	0	The trigger slope can be either rising or falling.
TRIGSLOPE_RISING	1	The trigger slope must be rising when the signal crosses the trigger value.
TRIGSLOPE_FALLING	2	The trigger slope must be falling when the signal crosses the trigger value.

See Also

API function `xPCScopes.ScopeGetTriggerSlopeStr`

xPCScopes.ScopeGetTriggerSlopeStr

Purpose Get trigger slope as string

Prototype `BSTR ScopeGetTriggerSlopeStr(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return The `xPCScopes.ScopeGetTriggerSlopeStr` method returns a string containing the trigger slope string.

Description The `xPCScopes.ScopeGetTriggerSlopeStr` method gets the trigger slope string for scope *scNum*. This method returns one of the following strings:

String	Description
`Either`	The trigger slope can be either rising or falling.
`Rising`	The trigger slope must be rising when the signal crosses the trigger value.
`Falling`	The trigger slope must be falling when the signal crosses the trigger value.

See Also API function `xPCScopes.ScopeGetTriggerSlope`

Purpose Get type of scope

Prototype `BSTR ScopeGetType(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return The `xPCScopes.ScopeGetType` method returns the scope type as a string. If there is an error, this function returns `-1`.

Description The `xPCScopes.ScopeGetType` method gets the type of scope *scNum*. This method returns one of the following strings:

String	Description
`HOST`	Scope of type host
`Target`	Scope of type target

xPCScopes.ScopeRemSignal

Purpose Remove signal from scope

Prototype `long ScopeRemSignal(long scNum, long sigNum);`

Member Of XPCAPICOMLIB.xPCScopes

Arguments

[in] *scNum*	Enter the scope number.
[in] *sigNum*	Enter a signal number.

Return If there is an error, this method returns 0. The xPCScopes.ScopeRemSignal method returns -1 upon success.

Description The xPCScopes.ScopeRemSignal method removes a signal from the scope with number *scNum*. The scope must already exist, and signal number *sigNum* must exist in the scope. Use xPCScopes.GetScopes to determine the existing scopes, and use xPCScopes.ScopeGetSignals to determine the existing signals for a scope. Use this function only when the scope is stopped. Use xPCScopes.ScopeGetState to check the state of the scope.

Purpose Set decimation of scope

Prototype `long ScopeSetDecimation(long scNum, long decimation);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
[in] *decimation*	Enter an integer for the decimation.

Return If there is an error, this method returns `0`. The `xPCScopes.ScopeSetDecimation` method returns `-1` upon success.

Description The `xPCScopes.ScopeSetDecimation` method sets the *decimation* of scope *scNum*. The decimation is a number, `N`, meaning every `N`th sample is acquired in a scope window. Use this function only when the scope is stopped. Use `xPCScopes.ScopeGetState` to check the state of the scope.

xPCScopes.ScopeSetNumPrePostSamples

Purpose Set number of pre- or posttriggering samples before triggering scope

Prototype

```
long ScopeSetNumPrePostSamples(long scNum, long prepost);
```

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
[in] *prepost*	A negative number means pretriggering, while a positive number means posttriggering. This function can only be used when the scope is stopped.

Return If there is an error, this method returns `0`. The `xPCScopes.ScopeSetNumPrePostSamples` method returns `-1` upon success.

Description The `xPCScopes.ScopeSetNumPrePostSamples` method sets the number of samples for pre- or posttriggering for scope *scNum* to *prepost*. Use this method only when the scope is stopped. Use `xPCScopes.ScopeGetState` to check the state of the scope. Use the `xPCScopes.GetScopes` method to get a list of scope numbers.

xPCScopes.ScopeSetNumSamples

Purpose

Set number of samples in one data acquisition cycle

Prototype

`long ScopeSetNumSamples(long scNum, long samples);`

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
[in] *samples*	Enter the number of samples you want to acquire in one cycle.

Return

If there is an error, this method returns `0`. The `xPCScopes.ScopesSetNumSamples` method returns `-1` upon success.

Description

The `xPCScopes.ScopeSetNumSamples` method sets the number of samples for scope *scNum* to *samples*. Use this function only when the scope is stopped. Use `xPCScopes.ScopeGetState` to check the state of the scope.

xPCScopes.ScopeSetTriggerLevel

Purpose

Set trigger level for scope

Prototype

```
long ScopeSetTriggerLevel(long scNum, double level);
```

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
[in] *level*	Value for a signal to trigger data acquisition with a scope.

Return

If there is an error, this method returns `0`. The `xPCScopes.ScopeSetTriggerLevel` method returns `-1` upon success.

Description

The `xPCScopes.ScopeSetTriggerLevel` method sets the trigger level *level* for scope *scNum*. Use this function only when the scope is stopped. Use `xPCScopes.ScopeGetState` to check the state of the scope.

Purpose Set trigger mode of scope

Prototype `long ScopeSetTriggerMode(long scNum, long triggermode);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
[in] *triggermode*	Trigger mode for a scope.

Return If there is an error, this method returns `0`. The `xPCScopes.ScopeSetTriggerMode` method returns `-1` upon success.

Description The `xPCScopes.ScopeSetTriggerMode` method sets the trigger mode of scope *scNum* to *triggermode*. Use this method only when the scope is stopped. Use `xPCScopes.ScopeGetState` to check the state of the scope. Use the `xPCScopes.GetScopes` method to get a list of scopes.

Use the constants defined here to interpret the trigger mode:

Constant	Value	Description
TRIGMD_FREERUN	0	The scope always triggers when it is ready to trigger, regardless of the circumstances. This is the default.

Constant	Value	Description
TRIGMD_SOFTWARE	1	Only a user can trigger the scope. It is always possible for a user to trigger the scope; however, if you set the scope to this trigger mode, user intervention is the only way to trigger the scope. No other triggering is possible.
TRIGMD_SIGNAL	2	Signal must cross a value before the scope is triggered.
TRIGMD_SCOPE	3	Scope is triggered by another scope at the trigger point of the triggering scope, modified by the value of `triggerscopesample` (see `scopedata`).

xPCScopes.ScopeSetTriggerSample

Purpose

Set sample number for triggering scope

Prototype

`long ScopeSetTriggerSample(long *scNum*, long *trigScSample*);`

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
[in] *trigScSample*	Enter a nonnegative integer for the number of samples acquired by the triggering scope before starting data acquisition on a second scope.

Return

If there is an error, this method returns `0`. The `xPCScopes.ScopeSetTriggerSample` method returns `-1` upon success.

Description

The `xPCScopes.ScopeSetTriggerSample` method sets the number of samples (*trigScSample*) a triggering scope acquires before it triggers a second scope (*scNum*). Use the `xPCScopes.GetScopes` method to get a list of scopes.

For meaningful results, set *trigScSample* between `-1` and (*nSamp*`-1`). *nSamp* is the number of samples in one data acquisition cycle for the triggering scope. However, no checking is done, and using a value that is too big causes the scope never to be triggered.

If you want to trigger a second scope at the end of a data acquisition cycle for the triggering scope, use a value of `-1` for *trigScSamp*.

xPCScopes.ScopeSetTriggerSignal

Purpose Select signal to trigger scope

Prototype `long ScopeSetTriggerSignal(long scNum, long triggerSignal);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
[in] *trigSignal*	Enter a signal number.

Return If there is an error, this method returns `0`. The `xPCScopes.ScopeSetTriggerSignal` method returns `-1` upon success.

Description The `xPCScopes.ScopeSetTriggerSignal` method sets the trigger signal of scope *scNum* to *trigSig*. The trigger signal *trigSig* must be one of the signals in the scope. Use this method only when the scope is stopped. You can use `xPCScopes.ScopeGetSignals` to get the list of signals in the scope. Use `xPCScopes.ScopeGetState` to check the state of the scope. Use the `xPCScopes.GetScopes` method to get a list of scopes.

Purpose

Set slope of signal that triggers scope

Prototype

```
long ScopeSetTriggerSlope(long scNum, long triggerslope);
```

Member Of

XPCAPICOMLIB.xPCScopes

Arguments

[in] *scNum*	Enter the scope number.
[in] *triggerSlope*	Enter the slope mode for the signal that triggers the scope.

Return

If there is an error, this method returns 0. The xPCScopes.ScopeSetTriggerSlope method returns -1 upon success.

Description

The xPCScopes.ScopeSetTriggerSlope method sets the trigger slope of scope *scNum* to *trigSlope*. Use this method only when the scope is stopped. Use xPCScopes.ScopeGetState to check the state of the scope. Use the xPCScopes.GetScopes method to get a list of scopes.

Use the constants defined here to set the trigger slope:

Constant	Value	Description
TRIGSLOPE_EITHER	0	The trigger slope can be either rising or falling.
TRIGSLOPE_RISING	1	The trigger signal value must be rising when it crosses the trigger value.
TRIGSLOPE_FALLING	2	The trigger signal value must be falling when it crosses the trigger value.

xPCScopes.ScopeSoftwareTrigger

Purpose Set software trigger of scope

Prototype `long ScopeSoftwareTrigger(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return If there is an error, this method returns `0`. The `xPCScopes.ScopeSoftwareTrigger` method returns `-1` upon success.

Description The `xPCScopes.ScopeSoftwareTrigger` method triggers scope *scNum*. The scope must be in the state `Waiting for trigger` for this method to succeed. Use `xPCScopes.ScopeGetState` to check the state of the scope. Use the `xPCScopes.GetScopes` method to get a list of scopes.

You can use the `xPCScopes.ScopeSoftwareTrigger` method to trigger the scope, regardless of the trigger mode.

Purpose

Start data acquisition for scope

Prototype

```
long ScopeStart(long scNum);
```

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return

If there is an error, this method returns `0`. The `xPCScopes.ScopeStart` method returns `-1` upon success.

Description

The `xPCScopes.ScopeStart` method starts or restarts the data acquisition of scope *scNum*. If the scope does not have to preacquire any samples, it enters the `Waiting for Trigger` state. The scope must be in state `Waiting to Start`, `Finished`, or `Interrupted` for this function to succeed. Call `xPCScopes.ScopeGetState` to check the state of the scope or, for host scopes that are already started, call `xPCScopes.IsScopeFinished`. Use the `xPCScopes.GetScopes` method to get a list of scopes.

xPCScopes.ScopeStop

Purpose Stop data acquisition for scope

Prototype `long ScopeStop(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return If there is an error, this method returns `0`. The `xPCScopes.ScopeStop` method returns `-1` upon success.

Description The `xPCScopes.ScopeStop` method stops the scope *scNum*. This sets the scope to the `Interrupted` state. The scope must be running for this function to succeed. Use `xPCScopes.ScopeGetState` to determine the state of the scope. Use the `xPCScopes.GetScopes` method to get a list of scopes.

Purpose

Get status of grid line for particular scope

Prototype

`long TargetScopeGetGrid(long scNum);`

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return

The `xPCScopes.TargetScopeGetGrid` method returns the state of the grid lines for scope *scNum*. If there is an error, this method returns `-1`.

Description

The `xPCScopes.TargetScopeGetGrid` method gets the state of the grid lines for scope *scNum* (which must be of type `SCTYPE_TARGET`). A return value of `1` implies grid on, while `0` implies grid off. Note that when the scope mode (as set or retrieved by `xPCGetScopes/xPCScopes.TargetScopeSetMode`) is set to `SCMODE_NUMERICAL`, the grid is not drawn even when the `grid` mode is set to `1`. Use the `xPCScopes.GetScopes` method to get a list of scopes.

xPCScopes.TargetScopeGetMode

Purpose Get scope mode for displaying signals

Prototype

```
long TargetScopeGetMode(long scNum);
```

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return The `xPCScopes.TargetScopeGetMode` method returns the value corresponding to the scope mode. The possible values are

- `SCMODE_NUMERICAL = 0`
- `SCMODE_REDRAW = 1`
- `SCMODE_SLIDING = 2`
- `SCMODE_ROLLING = 3`

If there is an error, this method returns `-1`.

Description The `xPCScopes.TargetScopeGetMode` method gets the mode of the scope *scNum*, which must be of type `SCTYPE_TARGET`. Use the `xPCScopes.GetScopes` method to get a list of scopes.

See Also API function `xPCScopes.TargetScopeGetModeStr`

Purpose

Get scope mode string for displaying signals

Prototype

```
BSTR TargetScopeGetModeStr(long scNum);
```

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return

The `xPCScopes.TargetScopeGetModeStr` method returns the string corresponding to the scope mode. The possible strings are

- `Numerical`
- `Redraw`
- `Sliding`
- `Rolling`

Description

The `xPCScopes.TargetScopeGetModeStr` method gets the mode string of the scope *scNum*, which must be of type `SCTYPE_TARGET`. Use the `xPCScopes.GetScopes` method to get a list of scopes.

See Also

API function `xPCGetScopes`

xPCScopes.TargetScopeGetViewMode

Purpose

Get view mode for target PC display

Prototype

```
long TargetScopeGetViewMode();
```

Member Of

`XPCAPICOMLIB.xPCScopes`

Return

The `xPCScopes.TargetScopeGetViewMode` method returns the view mode for the target PC screen. If there is an error, this method returns `-1`.

Description

The `xPCScopes.TargetScopeGetViewMode` method gets the view (zoom) mode for the target PC display. If the returned value is not zero, the number is of the scope currently displayed on the screen. If the value is `0`, then all defined scopes are currently displayed on the target PC screen. In the latter case, no scopes are in focus (that is, all scopes are unzoomed).

Purpose Get *y*-axis limits for scope

Prototype `VARIANT TargetScopeGetYLimits(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return The `xPCScopes.TargetScopeGetYLimits` method returns the upper and lower limits for scopes of type target.

Description The `xPCScopes.TargetScopeGetYLimits` method gets and copies the upper and lower limits for a scope of type `SCTYPE_TARGET` and with scope number *scNum*. If both elements are zero, the limits are autoscaled. Use the `xPCScopes.GetScopes` method to get a list of scopes.

xPCScopes.TargetScopeSetGrid

Purpose Set grid mode for scope

Prototype

```
long TargetScopeSetGrid(long scNum, long gridonoff);
```

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
[in] *gridonoff*	Enter a grid value (0 or 1).

Return If there is an error, this method returns 0. The `xPCScopes.TargetScopeSetGrid` method returns -1 upon success.

Description The `xPCScopes.TargetScopeSetGrid` method sets the grid of a scope of type `SCTYPE_TARGET` and scope number *scNum* to *gridonoff*. If *gridonoff* is 0, the grid is off. If *gridonoff* is 1, the grid is on and grid lines are drawn on the scope window. When the drawing mode of scope *scNum* is set to `SCMODE_NUMERICAL`, the grid is not drawn even when the grid mode is set to 1. Use the `xPCScopes.GetScopes` method to get a list of scopes.

xPCScopes.TargetScopeSetMode

Purpose

Set display mode for scope

Prototype

```
long TargetScopeSetMode(long scNum, long mode);
```

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.
in] *mode*	Enter the value for the mode.

Return

If there is an error, this method returns `0`. The `xPCScopes.TargetScopeSetMode` method returns `-1` upon success.

Description

The `xPCScopes.TargetScopeSetMode` method sets the mode of a scope of type `SCTYPE_TARGET` and scope number *scNum* to *mode*. You can use one of the following constants for *mode*:

- `SCMODE_NUMERICAL = 0`
- `SCMODE_REDRAW = 1`
- `SCMODE_SLIDING = 2`
- `SCMODE_ROLLING = 3`

Use the `xPCScopes.GetScopes` method to get a list of scopes.

xPCScopes.TargetScopeSetViewMode

Purpose Set view mode for scope

Prototype `long TargetScopeSetViewMode(long scNum);`

Member Of `XPCAPICOMLIB.xPCScopes`

Arguments

[in] *scNum*	Enter the scope number.

Return If there is an error, this method returns `0`. The `xPCScopes.TargetScopeSetViewMode` method returns `-1` upon success.

Description The `xPCScopes.TargetScopeSetViewMode` method sets the target PC screen to display one scope with scope number *scNum*. If you set *scNum* to `0`, the target PC screen displays all the scopes. Use the `xPCScopes.GetScopes` method to get a list of scopes.

Purpose

Set *y*-axis limits for scope

Prototype

```
long TargetScopeSetYLimits(long scNum, SAFEARRAY(double)*
Ylimitarray);
```

Member Of

`XPCAPICOMLIB.xPCScopes`

Arguments

[in] *`scNum`*	Enter the scope number.
[in, out] *`Ylimitarray`*	Enter a two-element array.

Return

If there is an error, this method returns `0`. The `xPCScopes.TargetScopeSetYLimits` method returns `-1` upon success.

Description

The `xPCScopes.TargetScopeSetYLimits` method sets the *y*-axis limits for a scope with scope number *`scNum`* and type `SCTYPE_TARGET` to the values in the double array *`YlimitArray`*. The first element is the lower limit, and the second element is the upper limit. Set both limits to `0.0` to specify autoscaling. Use the `xPCScopes.GetScopes` method to get a list of scopes.

xPCScRemSignal

Purpose

Remove signal from scope

Prototype

```
void xPCScRemSignal(int port, int scNum, int sigNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
sigNum	Enter a signal number.

Description

The `xPCScRemSignal` function removes a signal from the scope with number *scNum*. The scope must already exist, and signal number *sigNum* must exist in the scope. Use `xPCGetScopes` to determine the existing scopes, and use `xPCScGetSignals` to determine the existing signals for a scope. Use this function only when the scope is stopped. Use `xPCScGetState` to check the state of the scope. Use the `xPCGetScope` function to get the scope number.

See Also

API functions `xPCScAddSignal`, `xPCAddScope`, `xPCRemScope`, `xPCGetScopes`, `xPCScGetSignals`, `xPCScGetState`

Scope object method `remsignal`

Purpose

Set decimation of scope

Prototype

```
void xPCScSetDecimation(int port, int scNum,
int decimation);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
decimation	Enter an integer for the decimation.

Description

The `xPCScSetDecimation` function sets the *decimation* of scope *scNum*. The decimation is a number, `N`, meaning every `N`th sample is acquired in a scope window. Use this function only when the scope is stopped. Use `xPCScGetState` to check the state of the scope. Use the `xPCGetScope` function to get the scope number.

See Also

API functions `xPCScGetDecimation`, `xPCScGetState`

Scope object property `Decimation`

xPCScSetNumPrePostSamples

Purpose

Set number of pre- or posttriggering samples before triggering scope

Prototype

```
void xPCScSetNumPrePostSamples(int port, int
scNum, int prepost);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
prepost	A negative number means pretriggering, while a positive number means posttriggering. This function can only be used when the scope is stopped.

Description

The `xPCScSetNumPrePostSamples` function sets the number of samples for pre- or posttriggering for scope *scNum* to *prepost*. Use this function only when the scope is stopped. Use `xPCScGetState` to check the state of the scope. Use the `xPCGetScope` function to get the scope number.

See Also

API functions `xPCScGetNumPrePostSamples`, `xPCScGetState`

Scope object property `NumPrePostSamples`

xPCScSetNumSamples

Purpose

Set number of samples in one data acquisition cycle

Prototype

```
void xPCScSetNumSamples(int port, int scNum, int samples);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
samples	Enter the number of samples you want to acquire in one cycle.

Description

The `xPCScSetNumSamples` function sets the number of samples for scope *scNum* to *samples*. Use this function only when the scope is stopped. Use `xPCScGetState` to check the state of the scope. Use the `xPCGetScope` function to get the scope number.

See Also

API functions `xPCScGetNumSamples`, `xPCScGetState`

Scope object property `NumSamples`

xPCScSetTriggerLevel

Purpose

Set trigger level for scope

Prototype

```
void xPCScSetTriggerLevel(int port, int scNum,
double level);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
level	Value for a signal to trigger data acquisition with a scope.

Description

The `xPCScSetTriggerLevel` function sets the trigger level *level* for scope *scNum*. Use this function only when the scope is stopped. Use `xPCScGetState` to check the state of the scope. Use the `xPCGetScope` function to get the scope number for the trigger scope.

See Also

API functions `xPCScGetTriggerLevel`, `xPCScSetTriggerSlope`, `xPCScGetTriggerSlope`, `xPCScSetTriggerSignal`, `xPCScGetTriggerSignal`, `xPCScSetTriggerScope`, `xPCScGetTriggerScope`, `xPCScSetTriggerMode`, `xPCScGetTriggerMode`, `xPCScGetState`

Scope object property `TriggerLevel`

Purpose

Set trigger mode of scope

Prototype

```
void xPCScSetTriggerMode(int port, int scNum, int mode);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.
scNum	Enter the scope number.
mode	Trigger mode for a scope.

Description

The xPCScSetTriggerMode function sets the trigger mode of scope *scNum* to *mode*. Use this function only when the scope is stopped. Use xPCScGetState to check the state of the scope. Use the xPCGetScopes function to get a list of scopes.

Use the constants defined in xpcapiconst.h to interpret the trigger mode:

Constant	Value	Description
TRIGMD_FREERUN	0	The scope always triggers when it is ready to trigger, regardless of the circumstances. This is the default.
TRIGMD_SOFTWARE	1	Only a user can trigger the scope. It is always possible for a user to trigger the scope; however, if you set the scope to this trigger mode, user intervention is the only way to trigger the scope. No other triggering is possible.

Constant	Value	Description
TRIGMD_SIGNAL	2	Signal must cross a value before the scope is triggered.
TRIGMD_SCOPE	3	Scope is triggered by another scope at the trigger point of the triggering scope, modified by the value of triggerscopesample (see scopedata).

See Also

API functions xPCGetScopes, xPCScSetTriggerLevel, xPCScGetTriggerLevel, xPCScSetTriggerSlope, xPCScGetTriggerSlope, xPCScSetTriggerSignal, xPCScGetTriggerSignal, xPCScSetTriggerScope, xPCScGetTriggerScope, xPCScGetTriggerMode, xPCScGetState

Scope object method trigger

Scope object property TriggerMode

Purpose

Select scope to trigger another scope

Prototype

```
void xPCScSetTriggerScope(int port, int scNum,
int trigScope);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
trigScope	Enter the scope type to be triggered.

Description

The `xPCScSetTriggerScope` function sets the trigger scope of scope *scNum* to *trigScope*. This function can only be used when the scope is stopped. Use `xPCScGetState` to check the state of the scope. Use the `xPCGetScopes` function to get a list of scopes.

The scope type can be `SCTYPE_HOST` or `SCTYPE_TARGET`.

See Also

API functions `xPCGetScopes`, `xPCScSetTriggerLevel`, `xPCScGetTriggerLevel`, `xPCScSetTriggerSlope`, `xPCScGetTriggerSlope`, `xPCScSetTriggerSignal`, `xPCScGetTriggerSignal`, `xPCScGetTriggerScope`, `xPCScSetTriggerMode`, `xPCScGetTriggerMode`, `xPCScGetState`

Scope object property `TriggerScope`

xPCScSetTriggerScopeSample

Purpose

Set sample number for triggering scope

Prototype

```
void xPCScSetTriggerScopeSample(int port, int scNum, int
trigScSamp);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
trigScSamp	Enter a nonnegative integer for the number of samples acquired by the triggering scope before starting data acquisition on a second scope.

Description

The `xPCScSetTriggerScopeSample` function sets the number of samples (*trigScSamp*) a triggering scope acquires before it triggers a second scope (*scNum*). Use the `xPCGetScopes` function to get a list of scopes.

For meaningful results, set *trigScSamp* between `-1` and (*nSamp*`-1`). *nSamp* is the number of samples in one data acquisition cycle for the triggering scope. However, no checking is done, and using a value that is too big causes the scope never to be triggered.

If you want to trigger a second scope at the end of a data acquisition cycle for the triggering scope, enter a value of `-1` for *trigScSamp*.

See Also

API functions `xPCGetScopes`, `xPCScSetTriggerLevel`, `xPCScGetTriggerLevel`, `xPCScSetTriggerSlope`, `xPCScGetTriggerSlope`, `xPCScSetTriggerSignal`, `xPCScGetTriggerSignal`, `xPCScSetTriggerScope`, `xPCScGetTriggerScope`, `xPCScSetTriggerMode`, `xPCScGetTriggerMode`, `xPCScGetTriggerScopeSample`

Scope object properties `TriggerMode`, `TriggerSample`

xPCScSetTriggerSignal

Purpose

Select signal to trigger scope

Prototype

`void xPCScSetTriggerSignal(int` *`port`*`, int` *`scNum`*`, int` *`trigSig`*`);`

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
trigSig	Enter a signal number.

Description

The `xPCScSetTriggerSignal` function sets the trigger signal of scope *scNum* to *trigSig*. The trigger signal *trigSig* must be one of the signals in the scope. Use this function only when the scope is stopped. You can use `xPCScGetSignals` to get the list of signals in the scope. Use `xPCScGetState` to check the state of the scope. Use the `xPCGetScopes` function to get a list of scopes.

See Also

API functions `xPCGetScopes`, `xPCScGetState`, `xPCScSetTriggerLevel`, `xPCScGetTriggerLevel`, `xPCScSetTriggerSlope`, `xPCScGetTriggerSlope`, `xPCScGetTriggerSignal`, `xPCScSetTriggerScope`, `xPCScGetTriggerScope`, `xPCScSetTriggerMode`, `xPCScGetTriggerMode`

Scope object property `TriggerSignal`

xPCScSetTriggerSlope

Purpose

Set slope of signal that triggers scope

Prototype

```
void xPCScSetTriggerSlope(int port, int scNum,
int trigSlope);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.
scNum	Enter the scope number.
trigSlope	Enter the slope mode for the signal that triggers the scope.

Description

The xPCScSetTriggerSlope function sets the trigger slope of scope *scNum* to *trigSlope*. Use this function only when the scope is stopped. Use xPCScGetState to check the state of the scope. Use the xPCGetScopes function to get a list of scopes.

Use the constants defined in xpcapiconst.h to set the trigger slope:

Constant	Value	Description
TRIGSLOPE_EITHER	0	The trigger slope can be either rising or falling.
TRIGSLOPE_RISING	1	The trigger signal value must be rising when it crosses the trigger value.
TRIGSLOPE_FALLING	2	The trigger signal value must be falling when it crosses the trigger value.

See Also

API functions xPCGetScopes, xPCScSetTriggerLevel, xPCScGetTriggerLevel, xPCScGetTriggerSlope, xPCScSetTriggerSignal, xPCScGetTriggerSignal, xPCScSetTriggerScope, xPCScGetTriggerScope, xPCScSetTriggerMode, xPCScGetTriggerMode, xPCScGetState

Scope object property TriggerSlope

xPCScSoftwareTrigger

Purpose

Set software trigger of scope

Prototype

```
void xPCScSoftwareTrigger(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Description

The `xPCScSoftwareTrigger` function triggers scope *scNum*. The scope must be in the state `Waiting for trigger` for this function to succeed. Use `xPCScGetState` to check the state of the scope. Use the `xPCGetScopes` function to get a list of scopes.

You can use the `xPCScSoftwareTrigger` function to trigger the scope, regardless of the trigger mode.

See Also

API functions `xPCGetScopes`, `xPCScGetState`, `xPCIsScFinished`

Scope object method `trigger`

Scope object property `TriggerMode`

Purpose

Start data acquisition for scope

Prototype

```
void xPCScStart(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Description

The `xPCScStart` function starts or restarts the data acquisition of scope *scNum*. If the scope does not have to preacquire any samples, it enters the `Waiting for Trigger` state. The scope must be in state `Waiting to Start`, `Finished`, or `Interrupted` for this function to succeed. Call `xPCScGetState` to check the state of the scope or, for host scopes that are already started, call `xPCIsScFinished`. Use the `xPCGetScopes` function to get a list of scopes.

See Also

API functions `xPCGetScopes`, `xPCScGetState`, `xPCScStop`, `xPCIsScFinished`

Scope object method `start (scope object)`

xPCScStop

Purpose

Stop data acquisition for scope

Prototype

```
void xPCScStop(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Description

The `xPCScStop` function stops the scope *scNum*. This sets the scope to the `"Interrupted"` state. The scope must be running for this function to succeed. Use `xPCScGetState` to determine the state of the scope. Use the `xPCGetScopes` function to get a list of scopes.

See Also

API functions `xPCGetScopes`, `xPCScStart`, `xPCScGetState`

Scope object method `stop (scope object)`

Purpose Turn message display on or off

Prototype

```
void xPCSetEcho(int port, int mode);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
mode	Valid values are `0` Turns the display off `1` Turns the display on

Description On the target PC screen, the `xPCSetEcho` function sets the message display on the target PC on or off. You can change the mode only when the target application is stopped. When you turn the message display off, the message screen no longer updates.

See Also API function `xPCGetEcho`

xPCSetLastError

Purpose Set last error to specific string constant

Prototype

```
void xPCSetLastError(int error);
```

Arguments

error	Specify the string constant for the error.

Description The `xPCSetLastError` function sets the global error constant returned by `xPCGetLastError` to *error*. This is useful only to set the string constant to `ENOERR` to indicate no error was found.

See Also API functions `xPCGetLastError`, `xPCErrorMsg`

Purpose

Change initialization time-out value

Prototype

```
void xPCSetLoadTimeOut(int port, int timeOut);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
timeOut	Enter the new initialization time-out value.

Description

The `xPCSetLoadTimeOut` function changes the time-out value for initialization. The *timeOut* value is the time the function `xPCLoadApp` waits to check whether the model initialization for a new application is complete before returning. It enables you to set the number of initialization attempts to be made before signaling a time-out. When a new target application is loaded onto the target PC, the function `xPCLoadApp` waits for a certain time to check whether the model initialization is complete before returning. If the model initialization is incomplete within the allotted time, `xPCLoadApp` returns a time-out error.

By default, `xPCLoadApp` checks for target readiness five times, with each attempt taking approximately 1 second (less if the target is ready). However, in the case of larger models or models requiring longer initialization (for example, models with thermocouple boards), the default of about 5 seconds might be insufficient and a spurious time-out can be generated.

See Also

API functions `xPCGetLoadTimeOut`, `xPCLoadApp`, `xPCUnloadApp`

xPCSetLogMode

Purpose

Set logging mode and increment value of scope

Prototype

```
void xPCSetLogMode(int port, lgmode logging_data);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
logging_data	Logging mode and increment value.

Description

The `xPCSetLogMode` function sets the logging mode and increment to the values set in *logging_data*. See the structure `lgmode` for more details.

See Also

API function `xPCGetLogMode`

API structure `lgmode`

Target object property `LogMode`

Purpose Change value of parameter

Prototype

```
void xPCSetParam(int port, int paramIdx, const
double *paramValue);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
paramIdx	Parameter index.
paramValue	Vector with at least the correct size.

Description The `xPCSetParam` function sets the parameter *paramIdx* to the value in *paramValue*. For matrices, *paramValue* should be a vector representation of the matrix in column-major format. Although *paramValue* is a vector of doubles, the function converts the values to the correct types (using truncation) before setting them.

See Also API functions `xPCGetParamDims`, `xPCGetParamIdx`, `xPCGetParam`

xPCSetSampleTime

Purpose

Change target application sample time

Prototype

```
void xPCSetSampleTime(int port, double ts);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
ts	Sample time for the target application.

Description

The `xPCSetSampleTime` function sets the sample time, in seconds, of the target application to *ts*. Use this function only when the application is stopped.

See Also

API function `xPCGetSampleTime`

Target object property `SampleTime`

xPCSetScope

Purpose

Set properties of scope

Prototype

```
void xPCSetScope(int port, scopedata state);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
state	Enter a structure of type `scopedata`.

Description

The `xPCSetScope` function sets the properties of a scope using a *state* structure of type `scopedata`. Ensure that this structure contains the properties you want to set for the scope. You can set several properties at the same time. For convenience, call the function `xPCGetScope` first to populate the structure with the current values. You can then change the desired values. Use this function only when the scope is stopped. Use `xPCScGetState` to determine the state of the scope.

See Also

API functions `xPCGetScope`, `xPCScGetState`, `scopedata`

Scope object method `set (scope object)`

xPCSetStopTime

Purpose

Change target application stop time

Prototype

```
void xPCSetStopTime(int port, double tfinal);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
tfinal	Enter the stop time, in seconds.

Description

The `xPCSetStopTime` function sets the stop time of the target application to the value in *tfinal*. The target application will run for this number of seconds before stopping. Set *tfinal* to `-1.0` to set the stop time to infinity.

See Also

API function `xPCGetStopTime`

Target object property `StopTime`

Purpose Start target application

Prototype

```
void xPCStartApp(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Description The `xPCStartApp` function starts the target application loaded on the target machine.

See Also API function `xPCStopApp`

Target object method `start (target application object)`

xPCStopApp

Purpose

Stop target application

Prototype

```
void xPCStopApp(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Description

The `xPCStopApp` function stops the target application loaded on the target PC. The target application remains loaded, and all parameter changes made remain intact. If you want to stop and unload an application, use `xPCUnloadApp`.

See Also

API functions `xPCStartApp`, `xPCUnloadApp`

Target object method `stop (target application object)`

Purpose Get average task execution time

Prototype

```
double AverageTET();
```

Member Of `XPCAPICOMLib.xPCTarget`

Return The `xPCTarget.AverageTET` method returns the average task execution time (TET) for the target application. If there is an error, this method returns `-1`.

Description The `xPCTarget.AverageTET` method gets the TET for the target application. You can use this function when the target application is running or when it is stopped.

xPCTarget.GetAppName

Purpose Get target application name

Prototype

```
BSTR GetAppName();
```

Member Of `XPCAPICOMLib.xPCTarget`

Return The `xPCTarget.GetAppName` method returns a string with the name of the target application.

Description The `xPCTarget.GetAppName` method gets the name of the target application. You can use the return value, *`model_name`*, in a `printf` or similar statement. In case of error, the string is unchanged. Be sure to allocate enough space to accommodate the longest target name you have.

Purpose Get execution time for target application

Prototype `double GetExecTime();`

Member Of `XPCAPICOMLib.xPCTarget`

Return The `xPCTarget.GetExecTime` method returns the current execution time for a target application. If there is an error, this method returns `-1`.

Description The `xPCTarget.GetExecTime` method gets the current execution time for the running target application. If the target application is stopped, the value is the last running time when the target application was stopped. If the target application is running, the value is the current running time.

xPCTarget.GetNumOutputs

Purpose Get number of outputs

Prototype `long GetNumOutputs();`

Member Of `XPCAPICOMLib.xPCTarget`

Return The `xPCTarget.GetNumOutputs` method returns the number of outputs in the current target application.

Description The `xPCTarget.GetNumOutputs` method gets the number of outputs in the target application. The number of outputs equals the sum of the input signal widths of all output blocks at the root level of the Simulink model.

Purpose
Get number of tunable parameters

Prototype
```
long GetNumParams();
```

Member Of
`XPCAPICOMLib.xPCTarget`

Return
The `xPCTarget.GetNumParams` method returns the number of tunable parameters in the target application.

Description
The `xPCTarget.GetNumParams` method gets the number of tunable parameters in the target application. Use this method to see how many parameters you can get or modify.

xPCTarget.GetNumSignals

Purpose Get number of signals

Prototype `long GetNumSignals();`

Member Of `XPCAPICOMLib.xPCTarget`

Return The `xPCTarget.GetNumSignals` method returns the number of signals in the target application.

Description The `xPCTarget.GetNumSignals` method gets the total number of signals in the target application that can be monitored from the host. Use this function to see how many signals you can monitor.

Purpose Get number of states

Prototype `long GetNumStates();`

Member Of `XPCAPICOMLib.xPCTarget`

Return The `xPCTarget.GetNumStates` method returns the number of states in the target application.

Description The `xPCTarget.GetNumStates` method gets the number of states in the target application.

xPCTarget.GetOutputLog

Purpose

Copy output log data to array

Prototype

```
VARIANT GetOutputLog(long start, long numsamples,
long decimation,
long output_id);
```

Member Of

`XPCAPICOMLib.xPCTarget`

Arguments

[in] *start*	Enter the index of the first sample to copy.
[in] *numsamples*	Enter the number of samples to copy from the output log.
[in] *decimation*	Select whether to copy all the sample values or every `N`th value.
[in] *output_id*	Enter an output identification number.

Return

The `xPCTarget.GetOutputLog` method returns output log data. You get the data for each output signal. If there is an error, the method returns `VT_ERROR`, a scalar.

Description

The `xPCTarget.GetOutputLog` method gets the output log and copies that log to an array. Output IDs range from `0` to (`N`-1), where `N` is the return value of `xPCTarget.GetNumOutputs`. Entering `1` for *`decimation`* copies all values. Entering `N` copies every `N`th value.

For *`start`*, the sample indices range from `0` to (`N`-1), where `N` is the return value of `xPCTarget.NumLogSamples`. Get the maximum number of samples by calling the method `xPCTarget.NumLogSamples`.

Note that the target application must be stopped before you get the output log data.

xPCTarget.GetParam

Purpose Get parameter values

Prototype `VARIANT GetParam(long paramIdx);`

Member Of `XPCAPICOMLib.xPCTarget`

Arguments

[in] *paramIdx*	Enter the index for a parameter.

Return The `xPCTarget.GetParam` method returns the parameter values of a parameter.

Description The `xPCTarget.GetParam` method gets the parameter values of a parameter identified by *paramIdx*. This method returns an array of type `VARIANT` containing the parameter values, with the conversion of the values being done in column-major format. Each element in the array is a double, regardless of the data type of the actual parameter. You can query the dimensions of the array by calling the method `xPCTarget.GetParamDims`. See the Microsoft Visual Basic .NET 2003 `Demo` solution located in `C:\matlabroot\toolbox\rtw\targets\xpc\api\VBNET\SigsAndParamsDemo` for an example of how to use this method.

See Also API function `xPCTarget.GetParamDims`, `xPCTarget.SetParam`

Microsoft Visual Basic .NET 2003 demo solution located in `C:\matlabroot\toolbox\rtw\targets\xpc\api\VBNET\SigsAndParamsDemo`

xPCTarget.GetParamDims

Purpose

Get row and column dimensions of parameter

Prototype

`VARIANT GetParamDims(long paramIdx);`

Member Of

`XPCAPICOMLib.xPCTarget`

Arguments

[in] *paramIdx*	Parameter index.

Return

The `xPCTarget.GetParamDims` method returns a `VARIANT` array of two elements.

Description

The `xPCTarget.GetParamDims` method gets a `VARIANT` array of two elements. The first element contains the number of rows of the parameter, the second element contains the number of columns for your parameter.

Purpose Get parameter index

Prototype `long GetParamIdx(BSTR` *`blockName`*`, BSTR` *`paramName`*`);`

Member Of `XPCAPICOMLib.xPCTarget`

Arguments

[in] *`blockName`*	Enter the full block path generated by Real-Time Workshop.
[in] *`paramName`*	Enter the parameter name for a parameter associated with the block.

Return The `xPCTarget.GetParamIdx` method returns the parameter index for the parameter name. If there is an error, this function returns `-1`.

Description The `xPCTarget.GetParamIdx` method gets the parameter index for the parameter name (*`paramName`*) associated with a Simulink block (*`blockName`*). Both *`blockName`* and *`paramName`* must be identical to those generated at target application building time. The block names should be referenced from the file *`model_namept.m`* in the generated code, where *`model_name`* is the name of the model. Note that a block can have one or more parameters.

xPCTarget.GetParamName

Purpose

Get parameter name

Prototype

`VARIANT GetParamName(long paramIdx);`

Member Of

`XPCAPICOMLib.xPCTarget`

Arguments

[in] *paramIdx*	Enter a parameter index.

Return

The `xPCTarget.GetParamName` method returns a `VARIANT` array that contains two elements, the block path and parameter name, as strings.

Description

The `xPCTarget.GetParamName` method gets the parameter name and block name for a parameter with the index *paramIdx*. If *paramIdx* is invalid, `xPCGetLastError` returns nonzero, and the strings are unchanged. Get the parameter index with the method `xPCTarget.GetParamIdx`.

Purpose Get sample time

Prototype

```
double GetSampleTime();
```

Member Of `XPCAPICOMLib.xPCTarget`

Return The `xPCTarget.GetSampleTime` method returns the sample time, in seconds, of the target application. If there is an error, this function returns `-1`.

Description The `xPCTarget.GetSampleTime` method gets the sample time, in seconds, of the target application. You can get the error by using the method `xPCGetLastError`.

xPCTarget.GetSignal

Purpose

Get signal value

Prototype

```
double GetSignal(long sigNum);
```

Member Of

`XPCAPICOMLib.xPCTarget`

Arguments

[in] *`sigNum`*	Enter a signal number.

Return

The `xPCTarget.GetSignal` method returns the current value of signal *`sigNum`*.

Description

The `xPCTarget.GetSignal` method gets the current value of a signal. Use the `xPCTarget.GetSignalIdx` method to get the signal number.

xPCTarget.GetSignalidsfromLabel

Purpose

Get signal IDs from signal label

Prototype

`VARIANT GetSignalidsfromLabel(BSTR sigLabel);`

Member Of

`XPCAPICOMLib.xPCTarget`

Arguments

[in] *sigLabel*	Enter a signal label.

Return

The `GetSignalidsfromLabel` function returns a VARIANT array of the signal elements contained in the signal *sigLabel*.

Description

The`xPCTarget.GetSignalidsfromLabel` method returns a VARIANT array of the signal elements contained in the signal *sigLabel*.

This function assumes that you have labeled the signal for which you request the indices (see the **Signal name** parameter of the “Signal Properties Dialog Box” in the Simulink documentation). Note that xPC Target refers to Simulink signal names as signal labels. The creator of the application should already know the signal name/label.

See Also

API method `xPCTarget.GetSignalLabel`

xPCTarget.GetSignalLabel

Purpose Get signal label

Prototype

```
BSTR GetSignalLabel(long sigIdx);
```

Member Of `XPCAPICOMLib.xPCTarget`

Arguments

[in] *`sigIdx`*	Enter a signal index.

Return The `xPCTarget.GetSignalLabel` method returns the label of the signal.

Description The `xPCTarget.GetSignalLabel` method copies and gets the signal label of a signal with *`sigIdx`*. The method returns the signal label. This method assumes that you already know the signal index.

This function assumes that you have labeled the signal for which you request the indices (see the **Signal name** parameter of the “Signal Properties Dialog Box” in the Simulink documentation). Note that xPC Target refers to Simulink signal names as signal labels. The creator of the application should already know the signal name/label.

See Also API method `xPCTarget.GetSignalidsfromLabel`

Purpose Get signal index

Prototype

```
long GetSignalIdx(BSTR sigName);
```

Member Of `XPCAPICOMLib.xPCTarget`

Arguments

[in] *sigName*	Enter a signal name.

Return The `GetSignalIdx` function returns the index for the signal with name *`sigName`*.

Description The `xPCTarget.GetSignalIdx` method gets the index of a signal. The name must be identical to the name generated when the application was built. You should reference the name from the file *`model_namebio.m`* in the generated code, where *`model_name`* is the name of the model. The creator of the application should already know the signal name.

xPCTarget.GetSignalName

Purpose
Copy signal name to character array

Prototype
`BSTR GetSignalName(long *sigIdx*);`

Member Of
`XPCAPICOMLib.xPCTarget`

Arguments

[in] *`sigIdx`*	Enter a signal index.

Return
The `xPCTarget.GetSignalName` method returns the name of the signal.

Description
The `xPCTarget.GetSignalName` method copies and gets the signal name, including the block path, of a signal with *`sigIdx`*. The method returns a signal name, which makes it convenient to use in a `printf` or similar statement. This method assumes that you already know the signal index.

Purpose Get width of signal

Prototype `long GetSignalWidth(long sigIdx);`

Member Of `XPCAPICOMLib.xPCTarget`

Arguments

[in] *sigIdx*	Enter the index of a signal.

Return The `xPCTarget.GetSignalWidth` method returns the signal width for a signal with *sigIdx*. If there is an error, this function returns `-1`.

Description The `xPCTarget.GetSignalWidth` method gets the number of signals for a specified signal index. Although signals are manipulated as scalars, the width of the signal might be useful to reassemble the components into a vector. A signal's width is the number of signals in the vector.

xPCTarget.GetStateLog

Purpose

Get state log

Prototype

```
VARIANT GetStateLog(long start, long numsamples,
long decimation,
long state_id);
```

Member Of

`XPCAPICOMLib.xPCTarget`

Arguments

[in] *start*	Enter the index of the first sample to copy.
[in] *numsamples*	Enter the number of samples to copy from the output log.
[in] *decimation*	Select whether to copy all the sample values or every `N`th value.
[in] *state_id*	Enter a state identification number.
`[out, retval]` *Outarray*	The log is stored in *Outarray*, whose allocation is the responsibility of the caller.

Return

The `xPCTarget.GetStateLog` method returns the state log. If there is an error, the method returns `VT_ERROR`, a scalar.

Description

The `xPCTarget.GetStateLog` method gets the state log. You get the data for each state signal in turn by specifying the *state_id*. State IDs range from `1` to (`N-1`), where `N` is the return value of `xPCTarget.GetNumStates`. Entering `1` for *decimation* copies all values. Entering `N` copies every `N`th value. For *start*, the sample indices range from `0` to (`N-1`), where `N` is the return value of `xPCTarget.NumLogSamples`. Use the `xPCTarget.NumLogSamples` method to get the maximum number of samples.

Note that the target application must be stopped before you get the number.

Purpose

Get stop time

Prototype

```
double GetStopTime();
```

Member Of

`XPCAPICOMLib.xPCTarget`

Return

The `xPCTarget.GetStopTime` method returns the stop time as a double, in seconds, of the target application. If there is an error, this function returns `-1`.

Description

The `xPCTarget.GetStopTime` method gets the stop time, in seconds, of the target application. This is the amount of time the target application runs before stopping.

xPCTarget.GetTETLog

Purpose

Get TET log

Prototype

```
VARIANT GetTETLog(long start, long numsamples,
long decimation);
```

Member Of

`XPCAPICOMLib.xPCTarget`

Arguments

[in] *start*	Enter the index of the first sample to copy.
[in] *numsamples*	Enter the number of samples to copy from the TET log.
[in] *decimation*	Select whether to copy all the sample values or every `N`th value.
`[out, retval]` *Outarray*	The log is stored in *Outarray*, whose allocation is the responsibility of the caller.

Return

The `xPCTarget.GetTETLog` method returns the TET log. If there is an error, the method returns `VT_ERROR`, a scalar.

Description

The `xPCTarget.GetTETLog` method gets the task execution time (TET) log. Entering `1` for *decimation* copies all values. Entering `N` copies every `N`th value. For *start*, the sample indices range from `0` to (`N`-1), where `N` is the return value of `xPCTarget.NumLogSamples`. Use the `xPCTarget.NumLogSamples` method to get the maximum number of samples.

Note that the target application must be stopped before you get the number.

Purpose Get time log

Prototype

```
VARIANT GetTimeLog(long start, long numsamples,
long decimation);
```

Member Of XPCAPICOMLib.xPCTarget

Arguments

[in] *start*	Enter the index of the first sample to copy.
[in] *numsamples*	Enter the number of samples to copy from the time log.
[in] *decimation*	Select whether to copy all the sample values or every Nth value.

Return The xPCTarget.GetTimeLog method returns the time log. If there is an error, the method returns VT_ERROR, a scalar.

Description The xPCTarget.GetTimeLog method gets the time log. This is especially relevant in the case of value-equidistant logging, where the logged values are not necessarily uniformly spaced in time. Entering 1 for *decimation* copies all values. Entering N copies every Nth value. For *start*, the sample indices range from 0 to (N-1), where N is the return value of xPCTarget.NumLogSamples. Use the xPCTarget.NumLogSamples method to get the number of samples.

Note that the target application must be stopped before you get the number.

xPCTarget.GetxPCError

Purpose

Get error string

Prototype

```
BSTR GetxPCError();
```

Member Of

`XPCAPICOMLib.xPCTarget`

Return

The `xPCTarget.GetxPCError` method returns the string for the last reported error. If there is no error, this function returns 0.

Description

The `xPCTarget.GetxPCError` method gets the string of the error last reported by another COM API method. This value is reset every time you call a new method. Therefore, you should check this constant value immediately after a call to an API COM method. You can use this method in conjunction with the `xPCTarget.isxPCError` method, which detects that an error has occurred.

See Also

API function `xPCTarget.isxPCError`

Purpose Initialize target object to communicate with target PC

Prototype

```
long Init(IxPCProtocol* xPCProtocol);
```

Member Of `XPCAPICOMLib.xPCTarget`

Return If connection succeeds, this method returns `0`. Otherwise, there is no connection and the method returns `-1`.

Description The `xPCTarget.Init` method initializes the target object to communicate with the target PC referenced by the `xPCProtocol` object.

xPCTarget.IsAppRunning

Purpose Return running status for target application

Prototype

```
long IsAppRunning();
```

Member Of XPCAPICOMLib.xPCTarget

Return If the target application is stopped, the xPCTarget.IsAppRunning method returns 0. If the target application is running, this function returns 1. If there is an error, this method returns 0.

Description The xPCTarget.IsAppRunning method returns 1 or 0 depending on whether the target application is stopped or running.

Purpose

Return overload status for target PC

Prototype

```
long IsOverloaded();
```

Member Of

`XPCAPICOMLib.xPCTarget`

Return

If the application is running properly, the `xPCTarget.IsOverloaded` method returns `1`. If the CPU is overloaded, the `xPCTarget.IsOverloaded` function returns `0`. In case of error, this function returns `-1`.

Description

The `xPCTarget.IsOverloaded` method returns `1` if the target application is running properly and has not overloaded the CPU. It returns `0` if the target application has overloaded the target PC (CPU Overload).

xPCTarget.isxPCError

Purpose Return error status

Prototype

```
long isxPCError();
```

Member Of `XPCAPICOMLIB.xPCTarget`

Return The `xPCTarget.isxPCError` method returns the error status. If there is an error, this method returns `0`.

Description The `xPCTarget.isxPCError` method returns the error status. Use this method to check for any errors that might occur after a call to any of the `xPCTarget` class methods. If there is an error, call the `xPCTarget.GetxPCError` method to get the string for the error.

See Also API function `xPCTarget.GetxPCError`

Purpose

Load target application onto target PC

Prototype

`long LoadApp(BSTR pathstr, BSTR filename);`

Member Of

`XPCAPICOMLIB.xPCTarget`

Arguments

[in] *pathstr*	Enter the path to the target application file.
[in] *filename*	Enter the name of a compiled target application (`*.dlm`) without the file extension.

Return

If there is an error, this method returns `0`. The `xPCTarget.LoadApp` method returns `-1` upon success.

Description

The `xPCTarget.LoadApp` method loads the compiled target application to the target PC. *pathstr* must not contain the trailing backslash. *pathstr* can be set to NULL or to the string `'nopath'` if the application is in the current directory. The variable *filename* must not contain the target application extension.

Before returning, `xPCTarget.LoadApp` waits for a certain amount of time before checking whether the model initialization is complete. In the case where the model initialization is incomplete, `xPCTarget.LoadApp` returns a time-out error to indicate a connection problem (for example, `ETCPREAD`). By default, `xPCTarget.LoadApp` checks for target readiness five times, with each attempt taking approximately 1 second (less if the target is ready). However, in the case of larger models or models requiring longer initialization (for example, those with thermocouple boards), the default of about 5 seconds might be insufficient and a spurious time-out can be generated. The methods `xPCProtocol.GetLoadTimeOut` and `xPCProtocol.SetLoadTimeOut` control the number of attempts made.

xPCTarget.MaximumTET

Purpose

Copy maximum task execution time to array

Prototype

```
VARIANT MaximumTET();
```

Member Of

`XPCAPICOMLIB.xPCTarget`

Return

The `xPCTarget.MaximumTET` method returns a `VARIANT` object containing the maximum task execution time (TET) and the time at which the maximum TET was achieved. The maximum TET value is copied into the first element, and the time at which it was achieved is copied into the second element.

Description

The `xPCTarget.MaximumTET` method returns the maximum TET that was achieved during the previous target application run.

Purpose

Return maximum number of samples that can be in log buffer

Prototype

```
long MaxLogSamples();
```

Member Of

`XPCAPICOMLIB.xPCTarget`

Return

The `xPCTarget.MaxLogSamples` method returns the total number of samples. If there is an error, this function returns `-1`.

Description

The `xPCTarget.MaxLogSamples` method returns the total number of samples that can be returned in the logging buffers.

Note that the target application must be stopped before you get the number.

xPCTarget.MinimumTET

Purpose Copy minimum task execution time to array

Prototype `VARIANT MinimumTET();`

Member Of `XPCAPICOMLIB.xPCTarget`

Return The `xPCTarget.MinimumTET` method returns a `VARIANT` object containing the minimum task execution time (TET) and the time at which the minimum TET was achieved. The minimum TET value is copied into the first element, and the time at which it was achieved is copied into the second element.

Description The `xPCTarget.MinimumTET` method returns the minimum task execution time (TET) that was achieved during the previous target application run.

Purpose

Return number of samples in log buffer

Prototype

```
long NumLogSamples();
```

Member Of

`XPCAPICOMLIB.xPCTarget`

Return

The `xPCTarget.NumLogSamples` method returns the number of samples in the log buffer. If there is an error, this function returns `-1`.

Description

The `xPCTarget.NumLogSamples` method returns the number of samples in the log buffer. In contrast to `xPCTarget.MaxLogSamples`, which returns the maximum number of samples that can be logged (because of buffer size constraints), `xPCtarget.NumLogSamples` returns the number of samples actually logged.

Note that the target application must be stopped before you get the number.

xPCTarget.NumLogWraps

Purpose

Return number of times log buffer wraps

Prototype

```
long NumLogWraps();
```

Member Of

`XPCAPICOMLIB.xPCTarget`

Return

The `xPCTarget.NumLogWraps` method returns the number of times the log buffer wraps. If there is an error, this function returns `-1`.

Description

The `xPCTarget.NumLogWraps` method returns the number of times the log buffer wraps.

Note that the target application must be stopped before you get the number.

xPCTarget.SetParam

Purpose Change parameter value

Prototype

```
long SetParam(long paramIdx, SAFEARRAY(double)*
newparamVal);
```

Member Of XPCAPICOMLIB.xPCTarget

Arguments

[in] *paramIdx*	Parameter index.
[in, out] *newparamVal*	Vector with at least the correct size.

Return If there is an error, this method returns 0.

Description The xPCTarget.SetParam method sets the parameter *paramIdx* to the value in *newparamVal*. For matrices, *newparamVal* should be a vector representation of the matrix in column-major format. Although *newparamVal* is a vector of doubles, the function converts the values to the correct types (using truncation) before setting them.

See Also API functions xPCTarget.GetParam, xPCTarget.GetParamDims, xPCTarget.GetParamIdx

xPCTarget.SetSampleTime

Purpose

Change sample time for target application

Prototype

```
long SetSampleTime(double ts);
```

Member Of

`XPCAPICOMLIB.xPCTarget`

Arguments

[in] *`ts`*	Sample time for the target application.

Return

If there is an error, this method returns `0`. The `xPCTarget.SetSampleTime` method returns `-1` upon success.

Description

The `xPCTarget.SetSampleTime` method sets the sample time, in seconds, of the target application to *`ts`*. Use this method only when the application is stopped.

Purpose
Change stop time of target application

Prototype
`long SetStopTime(double tfinal);`

Member Of
`XPCAPICOMLIB.xPCTarget`

Arguments

[in] *`tfinal`*	Enter the stop time, in seconds.

Return
If there is an error, this method returns `0`. The `xPCTarget.SetStopTime` method returns `-1` upon success.

Description
The `xPCTarget.SetStopTime` method sets the stop time of the target application to the value in *`tfinal`*. The target application will run for this number of seconds before stopping. Set *`tfinal`* to `-1.0` to set the stop time to infinity.

xPCTarget.StartApp

Purpose Start target application

Prototype `long StartApp()`

Member Of `XPCAPICOMLIB.xPCTarget`

Return If there is an error, this method returns `0`. The `xPCTarget.StartApp` method returns `-1` upon success.

Description The `xPCTarget.StartApp` method starts the target application loaded on the target machine.

Purpose

Stop target application

Prototype

```
long StopApp();
```

Member Of

`XPCAPICOMLIB.xPCTarget`

Return

If there is an error, this method returns `0`. The `xPCTarget.StopApp` method returns `-1` upon success.

Description

The `xPCTarget.StopApp` method stops the target application loaded on the target PC. The target application remains loaded, and all parameter changes made remain intact. If you want to stop and unload an application, use `xPCTarget.UnLoadApp`.

xPCTarget.UnLoadApp

Purpose Unload target application

Prototype `long UnLoadApp();`

Member Of `XPCAPICOMLIB.xPCTarget`

Return If there is an error, this method returns `0`. The `xPCTarget.UnLoadApp` method returns `-1` upon success.

Description The `xPCTarget.UnloadApp` method stops the current target application, removes it from the target PC memory, and resets the target PC in preparation for receiving a new target application. The method `xPCTarget.LoadApp` calls this function before loading a new target application.

Purpose

Ping target PC

Prototype

```
int xPCTargetPing(int port);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.

Return

The xPCTargetPing function returns 1 if it successfully reaches the target. If there is an error, the function returns 0.

Description

The xPCTargetPing function pings the target PC and returns 1 or 0 depending on whether the target responds or not. This function returns an error string constant only when the input is incorrect (the port number is invalid or *port* is not open). All other errors, such as the inability to connect to the target, are ignored.

If you are using TCP/IP, note that xPCTargetPing will cause the target PC to close the TCP/IP connection. You can use xPCOpenConnection to reconnect. You can also use this xPCTargetPing feature to close the target PC connection in the event of an aborted TCP/IP connection (for example, if your host side program crashes).

See Also

API functions xPCOpenConnection, xPCOpenSerialPort, xPCOpenTcpIpPort, xPCClosePort

xPCTgScGetGrid

Purpose

Get status of grid line for particular scope

Prototype

```
int xPCTgScGetGrid(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Return

Returns the status of the grid for a scope of type `SCTYPE_TARGET`. If there is an error, this function returns `-1`.

Description

The `xPCTgScGetGrid` function gets the state of the grid lines for scope *scNum* (which must be of type `SCTYPE_TARGET`). A return value of `1` implies grid on, while `0` implies grid off. Note that when the scope mode (as set or retrieved by `xPCTgScGetMode`/`xPCTgScSetMode`) is set to `SCMODE_NUMERICAL`, the grid is not drawn even when the `grid` mode is set to `1`. Use the `xPCGetScopes` function to get a list of scopes.

See Also

API functions `xPCGetScopes`, `xPCTgScSetGrid`, `xPCTgScSetViewMode`, `xPCTgScGetViewMode`, `xPCTgScSetMode`, `xPCTgScGetMode`, `xPCTgScSetYLimits`, `xPCTgScGetYLimits`

Purpose Get scope mode for displaying signals

Prototype

```
int xPCTgScGetMode(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.
scNum	Enter the scope number.

Return The xPCTgScGetMode function returns the value corresponding to the scope mode. The possible values are

- SCMODE_NUMERICAL = 0
- SCMODE_REDRAW = 1
- SCMODE_SLIDING = 2
- SCMODE_ROLLING = 3

If there is an error, this function returns -1.

Description The xPCTgScGetMode function gets the mode (SCMODE_NUMERICAL, SCMODE_REDRAW, SCMODE_SLIDING, SCMODE_ROLLING) of the scope *scNum*, which must be of type SCTYPE_TARGET. Use the xPCGetScopes function to get a list of scopes.

See Also API functions xPCGetScopes, xPCTgScSetGrid, xPCTgScGetGrid, xPCTgScSetViewMode, xPCTgScGetViewMode, xPCTgScSetMode, xPCTgScSetYLimits, xPCTgScGetYLimits

Scope object property Mode

xPCTgScGetViewMode

Purpose Get view mode for target PC display

Prototype

```
int xPCTgScGetViewMode(int port);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.

Return The `xPCTgScGetViewMode` function returns the view mode for the target PC screen. If there is an error, this function returns `-1`.

Description The `xPCTgScGetViewMode` function gets the view (zoom) mode for the target PC display. If the returned value is not zero, the number is of the scope currently displayed on the screen. If the value is `0`, then all defined scopes are currently displayed on the target PC screen. In the latter case, no scopes are in focus (that is, all scopes are unzoomed).

See Also API functions `xPCGetScopes`, `xPCTgScSetGrid`, `xPCTgScGetGrid`, `xPCTgScSetViewMode`, `xPCTgScSetMode`, `xPCTgScGetMode`, `xPCTgScSetYLimits`, `xPCTgScGetYLimits`

Target object property `ViewMode`

xPCTgScGetYLimits

Purpose

Copy *y*-axis limits for scope to array

Prototype

```
void xPCTgScGetYLimits(int port, int scNum,
double *limits);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
limits	The first element of the array is the lower limit while the second element is the upper limit.

Description

The `xPCTgScGetYLimits` function gets and copies the upper and lower limits for a scope of type `SCTYPE_TARGET` and with scope number *scNum*. The limits are stored in the array *limits*. If both elements are zero, the limits are autoscaled. Use the `xPCGetScopes` function to get a list of scopes.

See Also

API functions `xPCGetScopes`, `xPCTgScSetGrid`, `xPCTgScGetGrid`, `xPCTgScSetViewMode`, `xPCTgScGetViewMode`, `xPCTgScSetMode`, `xPCTgScGetMode`, `xPCTgScSetYLimits`

Scope object property `YLimit`

xPCTgScSetGrid

Purpose Set grid mode for scope

Prototype

```
void xPCTgScSetGrid(int port, int scNum, int grid);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
grid	Enter a grid value.

Description The `xPCTgScSetGrid` function sets the grid of a scope of type `SCTYPE_TARGET` and scope number *scNum* to *grid*. If *grid* is `0`, the grid is off. If *grid* is `1`, the grid is on and grid lines are drawn on the scope window. When the drawing mode of scope *scNum* is set to `SCMODE_NUMERICAL`, the grid is not drawn even when the grid mode is set to `1`. Use the `xPCGetScopes` function to get a list of scopes.

See Also API functions `xPCGetScopes`, `xPCTgScGetGrid`, `xPCTgScSetViewMode`, `xPCTgScGetViewMode`, `xPCTgScSetMode`, `xPCTgScGetMode`, `xPCTgScSetYLimits`, `xPCTgScGetYLimits`

Scope object property `Grid`

Purpose

Set display mode for scope

Prototype

```
void xPCTgScSetMode(int port, int scNum, int mode);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.
scNum	Enter the scope number.
mode	Enter the value for the mode.

Description

The xPCTgScSetMode function sets the mode of a scope of type SCTYPE_TARGET and scope number *scNum* to *mode*. You can use one of the following constants for *mode*:

- SCMODE_NUMERICAL = 0
- SCMODE_REDRAW = 1
- SCMODE_SLIDING = 2
- SCMODE_ROLLING = 3

Use the xPCGetScopes function to get a list of scopes.

See Also

API functions xPCGetScopes, xPCTgScSetGrid, xPCTgScGetGrid, xPCTgScSetViewMode, xPCTgScGetViewMode, xPCTgScGetMode, xPCTgScSetYLimits, xPCTgScGetYLimits

Scope object property Mode

xPCTgScSetViewMode

Purpose

Set view mode for scope

Prototype

```
void xPCTgScSetViewMode(int port, int scNum);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.

Description

The `xPCTgScSetViewMode` function sets the target PC screen to display one scope with scope number *scNum*. If you set *scNum* to `0`, the target PC screen displays all the scopes. Use the `xPCGetScopes` function to get a list of scopes.

See Also

API functions `xPCGetScopes`, `xPCTgScSetGrid`, `xPCTgScGetGrid`, `xPCTgScGetViewMode`, `xPCTgScSetMode`, `xPCTgScGetMode`, `xPCTgScSetYLimits`, `xPCTgScGetYLimits`

Target object property `ViewMode`

Purpose

Set *y*-axis limits for scope

Prototype

```
void xPCTgScSetYLimits(int port, int scNum, const
double *Ylimits);
```

Arguments

port	Enter the value returned by either the function `xPCOpenSerialPort` or the function `xPCOpenTcpIpPort`.
scNum	Enter the scope number.
Ylimits	Enter a two-element array.

Description

The `xPCTgScSetYLimits` function sets the *y*-axis limits for a scope with scope number *scNum* and type `SCTYPE_TARGET` to the values in the double array *Ylimits*. The first element is the lower limit, and the second element is the upper limit. Set both limits to `0.0` to specify autoscaling. Use the `xPCGetScopes` function to get a list of scopes.

See Also

API functions `xPCGetScopes`, `xPCTgScSetGrid`, `xPCTgScGetGrid`, `xPCTgScSetViewMode`, `xPCTgScGetViewMode`, `xPCTgScSetMode`, `xPCTgScGetMode`, `xPCTgScGetYLimits`

Scope object property `Ylimit`

xPCUnloadApp

Purpose

Unload target application

Prototype

```
void xPCUnloadApp(int port);
```

Arguments

port	Enter the value returned by either the function xPCOpenSerialPort or the function xPCOpenTcpIpPort.

Description

The xPCUnloadApp function stops the current target application, removes it from the target PC memory, and resets the target PC in preparation for receiving a new target application. The function xPCLoadApp calls this function before loading a new target application.

See Also

API function xPCLoadApp

Target object methods load, unload

xPC Target C API Error Messages

Message	Description
ECOMPORTACCFAIL	COM port access failed
ECOMPORTISOPEN	COM port is already opened
ECOMPORTREAD	ReadFile failed while reading from COM port
ECOMPORTWRITE	WriteFile failed while writing to COM port
ECOMTIMEOUT	timeout while receiving: check serial link
EINVFILENAME	Invalid file name
EFILEOPEN	Error opening file
EFILEREAD	Error reading file
EFILERENAME	Error renaming file
EFILEWRITE	Error writing file
EINTERNAL	Internal Error
EINVADDR	Invalid IP Address
EINVALIDMODEL	Model name does not match saved value
EINVBAUDRATE	Invalid value for baudrate
EINVCOMMTYP	Invalid communication type
EINVCOMPORT	COM port can only be 0 or 1 (COM1 or COM2)
EINVLOGID	Invalid log identifier
EINVNUMPARAMS	Invalid number of parameters

Message	Description
EINVNUMSIGNALS	Invalid number of signals
EINVPARIDX	Invalid parameter index
EINVPORT	Invalid Port Number
EINVSCIDX	Invalid Scope Index
EINVSCTYPE	Invalid Scope type
EINVSIGIDX	Invalid Signal index
EINVTRIGMODE	Invalid trigger mode
EINVTRIGSLOPE	Invalid Trigger Slope Value
EINVTRSCIDX	Invalid Trigger Scope index
EINVARGUMENT	Invalid Argument
EINVDECIMATION	Decimation must be positive
EINVLGDATA	Invalid lgdata structure
EINVLGINCR	Invalid increment for value equidistant logging
EINVLGMODE	Invalid Logging mode
EINVNUMSAMP	Number of samples must be nonnegative
EINVSTARTVAL	Invalid value for "start"
EINVTFIN	Invalid value for TFinal
EINVTS	Invalid value for Ts (must be between 8e-6 and 10)
EINVWSVER	Invalid Winsock version (1.1 needed)
EINVXPCVERSION	Target has an invalid version of xPC Target
ELOADAPPFIRST	Load the application first
ELOGGINGDISABLED	Logging is disabled
EMALFORMED	Malformed message
EMEMALLOC	Memory allocation error
ENODATALOGGED	No data has been logged

Message	Description
ENOERR	No error
ENOFREEPORT	No free Port in C API
ENOMORECHANNELS	No more channels in scope
ENOSPACE	Space not allocated
EPARNOTFOUND	Parameter not found
EPARSIZMISMATCH	Parameter Size mismatch
EPINGCONNECT	Could not connect to Ping socket
EPINGPORTOPEN	Error opening Ping port
EPINGSOCKET	Ping socket error
EPORTCLOSED	Port is not open
ERUNSIMFIRST	Run simulation first
ESCTYPENOTTGT	Scope Type is not "Target"
ESIGNOTFOUND	Signal not found
ESOCKOPEN	Socket Open Error
ESTARTSIMFIRST	Start simulation first
ESTOPSCFIRST	Stop scope first
ESTOPSIMFIRST	Stop simulation first
ETCPCONNECT	TCP/IP Connect Error
ETCPREAD	TCP/IP Read Error
ETCPTIMEOUT	TCP/IP timeout while receiving data
ETCPWRITE	TCP/IP Write error
ETETLOGDISABLED	TET Logging is disabled
ETGTMEMALLOC	Target memory allocation failed
ETOOMANYSAMPLES	Too Many Samples requested
ETOOMANYSCOPES	Too many scopes are present
EUNLOADAPPFIRST	Unload the application first
EUSEDYNSCOPE	Use DYNAMIC_SCOPE flag at compile time

Message	Description
EWRITEFILE	LoadDLM: WriteFile Error
EWSINIT	WINSOCK: Initialization Error
EWSNOTREADY	Winsock not ready

Index

S

T

V

X