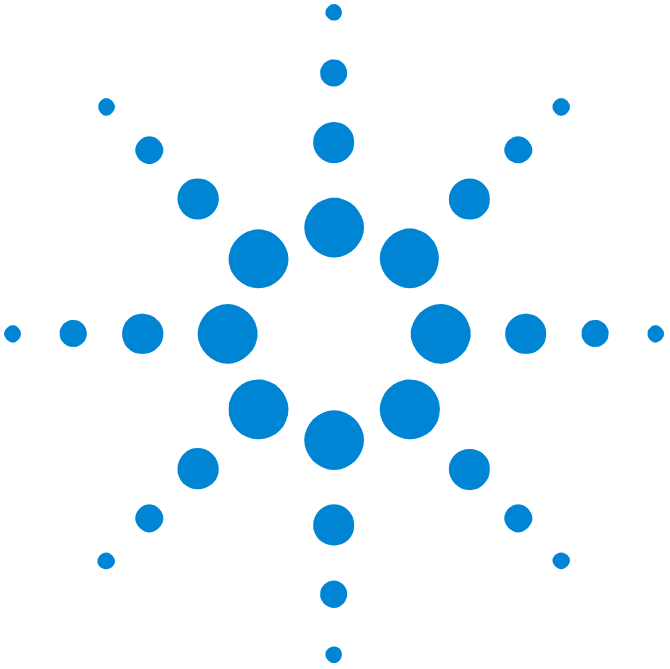


Agilent E2920 PCI-X Series Opt. 320 C-API/PPR

Programming Guide

Agilent Technologies

Important Notice

All information in this document is valid for Agilent E2929A, Agilent E2929B, Agilent E2922A and Agilent E2922B testcards.

© Agilent Technologies, Inc. 2002

Revision

June 2002

Printed in Germany

Agilent Technologies
Herrenberger Straße 130
D-71034 Böblingen
Germany

Authors: t3 medien GmbH

Warranty

The material contained in this document is provided "as is," and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied, with regard to this manual and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Agilent and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.

Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or subcontract, Software is delivered and licensed as "Commercial computer software" as defined in DFAR 252.227-7014 (June 1995), or as a "commercial item" as defined in FAR 2.101(a) or as "Restricted computer software" as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to Agilent Technologies' standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14 (June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

Safety Notices

CAUTION

A CAUTION notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a CAUTION notice until the indicated conditions are fully understood and met.

WARNING

A WARNING notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a WARNING notice until the indicated conditions are fully understood and met.

Trademarks

Windows NT ® and MS Windows ® are U.S. registered trademarks of Microsoft Corporation.

Contents

About This Guide 7

Documentation Overview 9

Programming Overview 11

- Programming Interfaces 12
- C Programming Libraries 13
- Generic C-API Functionality 14
- Protocol Permutation and Randomization Functionality 15
- Exception Handling 15
- Getting Started 17
  - How to Get Started 18
  - Example for Getting Started 20
- Benefits 21

Programming the Exerciser 23

- Reading From and Writing To the Memories 26
- Downloading Settings and Running the Exerciser 27
- Programming the Exerciser as a Requester-Initiator Device 28
  - Programming Generic Requester-Initiator Properties 29
  - Programming Requester-Initiator Block Transfers 30
  - Programming the Behavior of Block Transfers 35
- Programming the Exerciser as a Completer-Target Device 40
  - Programming a Target Decoder 41
  - Programming the Configuration Space 45
  - Programming the Completer-Target Behavior 48
  - Programming Generic Completer-Target Properties 52
  - Programming a Split Condition 53

Programming the Exerciser as a Completer-Initiator Device 55
Programming Generic Completer-Initiator Properties 55
Programming the Completer-Initiator Behavior 57
Programming the Exerciser as a Requester-Target Device 61
Programming Generic Requester-Target Properties 61
Programming a Split Completion Decoder 63
Programming the Requester-Target Behavior 63
Controlling the Exerciser 67
Scheduling Block Transfers and Split Completions 68
Programming the Data Generator 73
Programming Errors Injection 76
Programming the Expansion ROM 80
Programming the Data Memory 81
How to Program the Data Memory 83
Example for Programming the Data Memory 83
Programming Data Transfer To and From the Host 84
Example for Host Access 84
Programming PCI-X Interrupts 85
How to Generate PCI-X Interrupts 85
Example for Programming PCI-X Interrupts 87
Programming the Analyzer 89
Programming the Protocol Observer 90
How to Program the Protocol Observer 91
Example for Programming the Protocol Observer 93
Programming Pattern Terms 94
How to Program Pattern Terms 94
Example for Programming Pattern Terms 95
Programming the Trigger Sequencer 96
How to Program the Trigger Sequencer 99
Example for Programming the Trigger Sequencer 100
Programming the Trace Memory 104
How to Program the Trace Memory 105
Example for Programming the Trace Memory 106

Programming the Performance Sequencer 111
How to Program the Performance Sequencer 113
Example for Programming the Performance Sequencer 115

Programming Protocol Permutator and Randomizer Properties 117

Introduction 118
Contributions of the PCI-X PPR Software 120
Operation Principles 121
Generating Permutations 123
How to Write a Test Program 127
Example Test Design 128
Preparing for PPR Programming 131
How to Prepare for PPR Programming 132
Example for Preparing for PPR Programming 133
Programming Requester-Initiator Block Permutations 134
How to Program RI Block Permutations 139
Example for Programming RI Block Permutations 141
Programming RI Behavior Permutations 142
How to Program RI Behavior Permutations 144
Example for Programming RI Behavior Permutations 146
Programming CT Behavior Permutations 147
How to Program CT Behavior Permutations 148
Example for Programming CT Behavior Permutations 149
Programming CI Behavior Permutations 150
How to Program CI Behavior Permutations 151
Example for Programming CI Behavior Permutations 152
Programming RT Behavior Permutations 153
How to Program RT Behavior Permutations 154
Example for Programming RT Behavior Permutations 155
Generating PPR Reports 155
How to Generate PPR Reports 156
Example for Generating PPR Reports 157
Running a PPR Test 157
How to Run a PPR Test 157
Example for Running a PPR Test 157

Analyzing the Report 159
Report Header 159
Report of Block Permutations 160
Report of Requester-Initiator Behavior Permutation 167
Report of Requester-Initiator Block vs. Requester-Initiator Behavior Permutation 171
Further Options and Possibilities 172
Report Listing 174
Code Listing 185

Synchronizing the Environment 189

Card Status Reporting 191
How to Access the Card Status Register 192
Example for Accessing the Card Status Register 193
Generic Testcard Setup and Power-Up Control 194
How to Program Generic Testcard Properties and Power-Up Control 195
Programming the Mailbox 195
How to Program the Mailbox 198
Example for Programming the Mailbox 199
Programming the Trigger I/O 199
How to Program the Trigger I/O 200
Example for Programming the Trigger I/O 202
Programming the Display 203
Example for Programming the Display 204

About This Guide

Programming Interface

The Agilent E2920 PCI-X Series testcards are used for testing PCI-X chips, cards and systems. For this purpose, the testcard allows you to develop test programs by using:

- C-Application Programming Interface (C-API)

  The C-API allows you programmable control for the whole system and allows you the integration into existing test environments.

- Additional functions performed by the PCI-X Permutator and Randomizer software (PPR)

  These functions allow you to prepare and perform systematic functional tests at the protocol level, especially exposing PCI-X devices of a computer system to variable stressful PCI-X traffic.

Programming Guide Structure

For developing C programs or for using the command line interface of the graphical user interface, this Programmer's Guide gives you good background knowledge of the programming models for the Agilent E2920 PCI-X Series testcards.

The programming guide contains the following sections:

- *"Programming Overview" on page 11* gives basic information about writing C programs, such as where to find the required libraries, compilation and error checking.

  This section also provides information about the first steps to be performed in any C program, such as how the testcard is connected to the control PC and initialized.

- *"Programming the Analyzer" on page 89* provides information about programming models for all tasks of PCI-X analysis to monitor the PCI-X bus, to detect specific events, to measure and to evaluate the occurrences of signals on the bus.

- *"Programming the Exerciser" on page 23* provides information about the programming models for programming the testcard as a initiator and as a target device and for resources shared by both, such as data memory and compare unit.

- *"Programming Protocol Permutator and Randomizer Properties" on page 117* provides an overview of the features of the software, and shows how a test program is designed and implemented.
- *"Synchronizing the Environment" on page 189* provides information about the programming models for the available application interfaces, such as trigger I/O sequencer, LED display and mailbox.

Documentation Overview

This section shows you the different types of documents offered by Agilent Technologies and gives you an overview of which documents are available when you work with the Agilent E2929A/B PCI-X Exerciser and Analyzer.

All documents are valid for both Agilent E2929A and Agilent E2929B testcards. The following documents are available:

Getting Started Guide

- **Getting Started Guide**

  Introduces standard analysis features and provides an example of how to set up the protocol observer.

  This guide also gives detailed information about the hardware and interfaces.

User's Guides

- **Agilent E2929A/B Opt. 300 PCI-X Exerciser User's Guide**

  Provides information on programming the testcard as an initiator and/or target device. It shows you how to actively stimulate the PCI-X bus.

  This guide shows how to:

  – Initiate data transfers on the PCI-X bus (act as requester-initiator).

  – Act as completer-target.

  – Handle split completion transactions (act as completer-initiator).

  – Handle open requests (act as requester-target).

- **Agilent E2929A/B Opt. 100 PCI-X Analyzer User's Guide**

  Provides information on how to examine the behavior of a PCI-X device on the bus and shows how to perform functional tests such as data compares.

- **Agilent E2929A/B Opt. 200 PCI-X Performance Optimizer User's Guide**

  Provides all features that are needed to evaluate and optimize any device under test in terms of the performance.

- **Agilent E2920 PCI-X Series Opt. 320 C-API/PPR Programmer's Guide**

  Provides information on how to set up test programs using the C functions described in the corresponding C-API/PPR Reference.

GUI and C-API/PPR References

- **Agilent E2929A/B Windows and Dialog Boxes Reference**

  Provides reference information on all windows and dialog boxes of the Agilent E2920 graphical user interface (GUI).

- **Agilent E2929A/B Opt. 320 C-API/PPR Reference**

  Describes all C functions, types and definitions of the application programming interface of the Agilent E2929A/B PCI-X testcard.

  This reference also provides the commands and abbreviations that are used in the command line interface (CLI) of the graphical user interface.

- **Agilent E2922A/B Opt. 320 C-API/PPR Reference**

  Describes all C functions, types and definitions of the application programming interface of the Agilent E2922A/B PCI-X testcard.

  This reference also provides the commands and abbreviations that are used in the command line interface (CLI) of the graphical user interface.

Programming Overview

The following sections give basic information about the C-API and the PPR software:

- The ways in programming the testcard are shown in *"Programming Interfaces" on page 12*.
- Where to find the libraries, what you must do when writing C programs, and how to compile the programs depending on the operating system, can be found in *"C Programming Libraries" on page 13*.
- The features of the C-API and the PPR software can be found in *"Generic C-API Functionality" on page 14* and *"Protocol Permutation and Randomization Functionality" on page 15*.
- Error handling macros, which are needed to return error codes of C functions, are explained in *"Exception Handling" on page 15*.

Programming Interfaces

The testcard can be programmed in the following ways:

- By writing C programs

  The testcard is shipped with an application programming interface (the C-API) for the C programming language.

  See *"C Programming Libraries" on page 13*.

- By using the command line interface (CLI)

  The CLI provides an easy-to-use graphical user interface for entering commands. Descriptions of the CLI commands can be found in *the C-API/PPR Reference*, together with their corresponding C function.

  For more information, refer to "*Using the Command Line Interface*" in the *Agilent E2929A/B Opt. 300 PCI-X Exerciser User's Guide*.

- By writing TCL programs

  See `file <instdir>\PCIX\src\tcl\readme.txt`.

Hints for programming on 64 bit systems

If you plan to run the PCI-X software under 64 bit Itanium systems, you should read the following.

Targeted are currently the 64 bit Microsoft .NET Server OSes.

To install, you need a separate installation file, named setup64.exe, located in the CD's *ia64* directory. Do not install the 32bit setup.exe.

On 64bit Itanium systems the following is true:

- Kernel mode:

  Drivers always need to be 64 bit drivers; 32 bit drivers wont work. Especially, this means that you can't use the existing 32 bit drivers. Our 64 bit drivers are named *b_2kpci_64.sys*, *b_2khif_64.sys*, *b_usb_64.sys* and *b_usbgen_64.sys*.

- User mode:

  If you are starting an application, the .exe (and all needed dlls) need to be either all 32 bit files or all need to be 64 bit files, i.e. you cannot mix them. For example a 64 bit .exe cannot use a 32 bit dll.

  Our 64-bit dlls always have the suffix "xp64", e.g. capixp64.dll (instead of capikk.dll in 32 bit mode).

- The PCI-X GUI always only runs in 32bit mode (so they alway needs the corresponding 32 bit dlls).

  If you want to write your own C-API programs, you can use the provided 64bit dlls though and run your program as 64 bit executable (32 bit mode is forced only when using the GUI).

C Programming Libraries

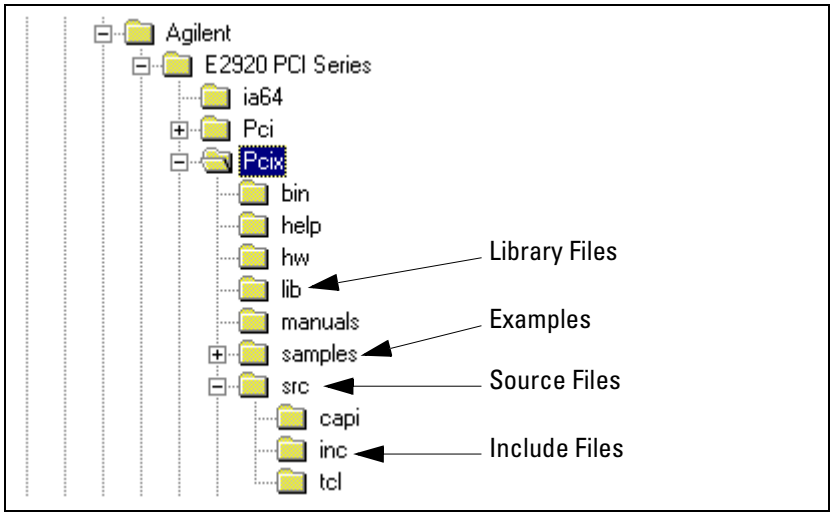
The Installation Wizard stores by default the library files, user documentation and programming examples to the PC. You can also develop your test program on a different PC (in the "Demo/Offline Mode" of the software) and later upload your application to the control PC.

Directory Structure

All required files are automatically installed with the control software and can be found in the subdirectories of the Agilent PCI-X Series home directory. The following figure shows the directory structure on a Windows NT system.

The home directory is:

```
C:\Program Files\Agilent\E2920 PCI(X) Series 1.4\PCIX\
```



When developing C programs for the testcard, you need to:

- Include only the header file `xpciapi.h` into your program, because it includes all necessary header files.

- Enter the paths to include files, library files, and/or source files into the directory settings of your developing environment.
  - For Windows 2000, you also have to include `xcapikk.lib` to the developing environment.
  - For Windows NT, you also have to include `xcapint.lib` to the developing environment.

Examples

Many ready-to-use example programs can be found in the **samples** directory. The **user documentation** for hardware, software, and options uses many of these examples to explain the functions.

Platform-Dependence

All sample programs can be compiled with Microsoft ® VC 6.0.

Communication with E2920 Series PCI-X testcards uses the E2920 Series C-API. The C-API is the interface for testcard communication. The C-API is available in binary form for a number of operating systems, and as compilable source code. A workspace for compiling the CAPI is available under `src\capi\capi.dsw`.

Generic C-API Functionality

The C-API is used to program all analyzer, exerciser and performance optimizer functionalities.

For all features of the testcard, refer to:

- *Agilent E2929A/B Getting Started Guide*
- *Agilent E2929A/B Opt. 300 PCI-X Exerciser User's Guide*
- *Agilent E2929A/B PCI-X Analyzer User's Guide*
- *Agilent E2929A/B Opt. 200 PCI-X Performance Optimizer User's Guide*

Protocol Permutation and Randomization Functionality

The PCI-X Protocol Permutation and Randomization software adds functions to the C-API for preparing and performing systematic functional tests at the protocol level, especially tests for exposing PCI-X devices of a computer system to variable stressful PCI-X traffic.

For more information on the PPR functionality, refer to *"Introduction" on page 118*.

Exception Handling

Try blocks are an efficient way of catching errors in a series of C-API function calls and are particularly useful for situations where some cleanup is required after an error occurs.

Error Checking

The `BX_TRY()` macro in the `BX_TRY_BEGIN {}` section checks if there was an error in the most recent call. If an error occurs, processing in the TRY block stops and the program proceeds with the `BX_TRY_CATCH {}` section.

All functions in TRY macros must return `bx_errtype`..

Error Handling

In case of an error, the easiest action is to print out the error string.

To get the error string, use the following function calls:

- `BestXLastErrorStringGet(handle)` if you know the handle (handle-based error checking).
- `BestXErrorStringGet(error number)` if you know the error number (non-handle-based error checking)

Example with Handle-Based Error Checking

The principle of exception handling is shown by means of the following example:

```
bx_errtype SomeFunction(bx_handletype handle)
{
   BX_TRY_VARS_NO_PROG;    /* declares all necessary variables

   BX_TRY_BEGIN            /* starts a try block
   {
      // API calls using the BX_TRY macro (returning bx_errtype)
      BX_TRY(BestXPing(handle));
      BX_TRY(BestXDisplayStringWrite(handle, "PCIX"));

   }
   BX_TRY_CATCH                /* starts the catch block
                               /* (optional)
   {
      // errorhandling (cleanup,ignore and/or handle error)
      // (optional)
      BX_TRY_RET=BX_E_OK;      /* ignores the error;
                               /* you can also switch over
                               /* BX_TRY_RET and react to
                               /* different errors.
      printf("%s\n", BestXLastErrorStringGet(handle));
   }
   BX_ERRETURN(BX_TRY_RET)      /* the reason for the failure
                                /* you can evaluate this macro
}
```

Non-Handle-Based Error Checking

The following functions do not provide handles, therefore they cannot be used with handle-based error checking methods:

- `BestXDevIdentifierGet()`
- `BestXPCICfgMailboxReceiveRegRead()`
- `BestXPCICfgMailboxSendRegWrite()`

Handle Initialization

The following function initializes the handle. The handle is valid only if this function returns the handle successfully:

- `BestXOpen()`

Calling the Macro

This macro can be called in the following way:

- `BX_TRY(BestDevIdentifierGet(vendor_id, device_id, number, &devid));`

For error codes, refer to "bx_errtype" in the *C-API/PPR Programming Reference*.

Getting Started

The first step in running tests on the testcard is to initialize the testcard and its connections. The testcard can be controlled via PCI-X port, RS-232 serial interface or Fast Host Interface.

Some typical initialization routines for each type of control connection are shown in *"Example for Getting Started" on page 20*.

PCI-X Port

The testcard communicates via the PCI-X bus through its configuration space.

No system resources are required to program the testcard. The PCI-X port is especially useful when the DUT controls the testcard in order to run parallel and synchronized tests. This port should not be used if changing the system by the testcard is not allowed (for example, for memory mapping).

RS-232 Serial Interface

The RS-232 serial interface of the Agilent E2929A/B testcard provides an easy-to-use control interface, which is available on all PCs and notebook computers. It can be run at 9600, 19200, 38400, and 57600 baud.

USB Port

The USB port of the Agilent E2929A/B testcard can be used to connect more than 4 testcards to one host without using PCI-X connections. With an USB hub, as many as 256 PCI-X testcards can be controlled simultaneously.

The first USB connection to a testcard to be found is assigned to "0", the second to "1", and so forth. The order in which the cards are found is not predictable. To see which card is connected to the session, use BestXPing and watch the LEDs (after the BestXOpen call).

Fast Host Interface Port

The Fast Host Interface port of the Agilent E2929A/B testcard provides an easy-to-install connection to the control PC with higher throughput than an RS-232 interface in both read and write directions.

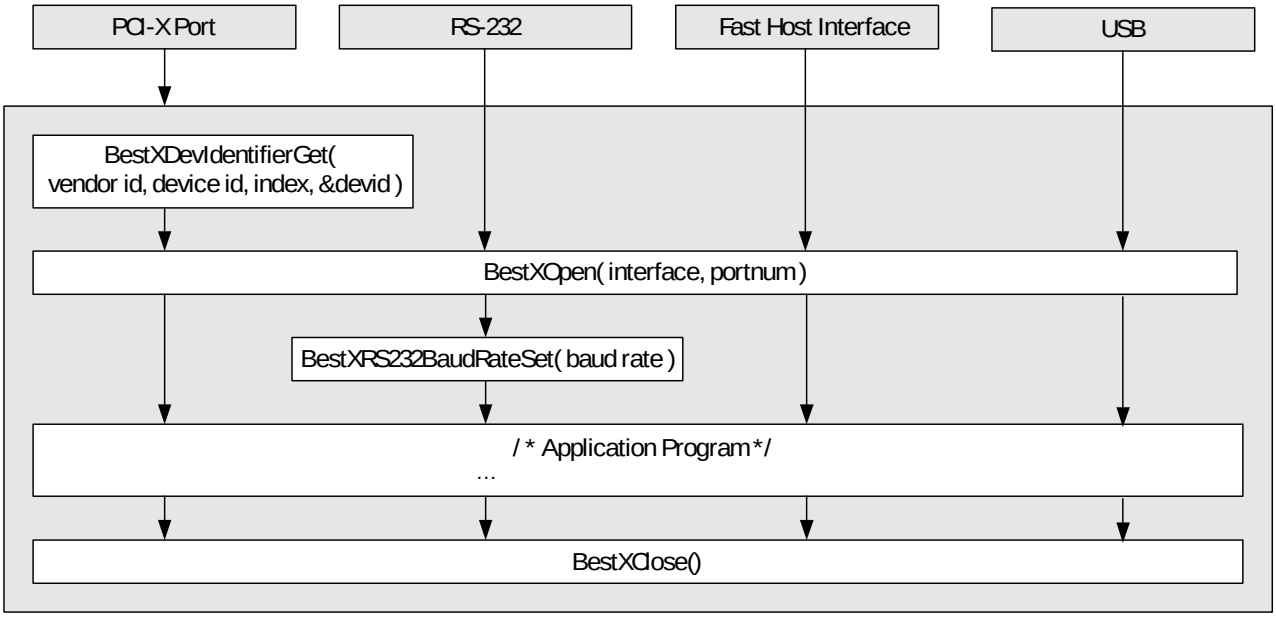
The control PC must be equipped with the Fast Host Interface card (delivered with the PCI-X Analyzer) and connected to the parallel port of the testcard.

Specification

Maximum transfer rate: 4 MB/s (using the Fast Host Interface of the Agilent E2929A/B testcard).

How to Get Started

The following figure shows which commands would be used to operate the testcard at different ports. This figure also shows the integration of these functions into the test program.



Programming Steps

The testcard is initialized as follows:

1 Open a session with *BestXOpen*. This call:

- Initializes the session data structures.
- Establishes the connection to the testcard and checks for valid card-type and revision.
- Returns the session handle on success.

NOTE If the **PCI-X Bus** is used as the controlling interface port, use *BestXDevIdentifierGet* to get the device number of the testcard, which is is used in *BestXOpen* for device identification.

If the **RS-232** serial interface is used, you have to set the baud rate with *BestXRS232BaudRateSet*.

2 After inserting the application code, close the session with *BestXClose*. This call:

- Frees the session data structures.
- Disconnects from the testcard.

Example for Getting Started

Task Write a C-API application that writes the text “Hello World” to the hexadecimal display on the E2929A card.

NOTE The following example can be used as framework for all further code fragments using the C-API in this document.

Implementation

```
#include "xpciapi.h"

int main(int argc, char* argv[])
{
   int i;

   BX_TRY_VARS_NO_PROG;

   /* Enter additional local variable declarations here */
   bx_handletype handle;

   BX_TRY_BEGIN
   {
      /* Open the communication session to testcard, initialize  */
      /* internal structures */

      BX_TRY(BestXOpen(&handle, BX_PORT_RS232, BX_PORT_COM1));

      /* If using RS232, set baud rate: */
      BX_TRY(BestXRS232BaudRateSet(handle, BX_BD_57600));

      /* Insert here your C-API calls */
      /* For example:*/
      /* Write “Hello World” to the display.*/

      for (i=0;i<10;i++)
      {
         BX_TRY (BestXDisplayStringWrite(handle, "HEL-"));
         BX_TRY (BestXDisplayStringWrite(handle, "HEL\\"));
         BX_TRY (BestXDisplayStringWrite(handle, "HEL|"));
         BX_TRY (BestXDisplayStringWrite(handle, "HEL/"));
      }

      /* Close the session to deallocate memory. */
      BX_TRY(BestXClose(handle));
   }

   BX_TRY_CATCH
   {
       printf(BestXErrorStringGet(BX_TRY_RET));
      /* cleanup, if necessary */
   }
   BX_ERRETURN(BX_TRY_RET);

}
```

NOTE This program can also be found under <INST_DIR>\ PCIX\Samples\HelloWorld.cpp.

Benefits

When setting up tests, you can take advantage of the following features of the Exerciser and Analyzer and the PCI-X Permutator and Randomizer software:

- **Creating controlled protocol corner cases**

  The software makes it possible to expose device or system under test to corner case traffic, to add system and parity errors, to assert and deassert signal lines and so on.

  Tests can be set up that add as many Exerciser and Analyzers as required and let them transfer data blocks repeatedly to generate enough traffic to stress the PCI-X system.

- **Data-integrity testing**

  The software makes it possible to use the Exerciser and Analyzer memory functions to comfortably write, read and compare data blocks.

- **Emulating typical peripheral traffic**

  The software makes it possible to substitute test devices with Exerciser and Analyzers. Testcards can be set up to behave like any device. The memory is programmable with any content. There is no need to exchange devices in the system for testing reasons to get “realistic” traffic.

  The PCI-X Permutator and Randomizer software intensifies the possibilities by systematically varying transfer parameters to examine protocol corner cases.

- **Storing and analyzing bus traffic**

  The software makes it possible to find illegal behavior on protocol and signal level using advanced listers (waveform viewer, bus cycle lister, transaction lister) of the (optional) graphical user interface of Exerciser and Analyzer. These listers allow a detailed analysis of all events that have occurred on the considered bus.

- **Stressing on multiple PCI/PCI-X buses**

  The software makes it possible to use multiple testcards to generate stress traffic from one bus system to another over PCI/PCI-X-to-PCI-/PCI-X bridges.

- **Deterministic and reproducible tests**

  In contrast to PCI-X traffic generated by other test devices, the generated variations are deterministic and reproducible. This guarantees coverage and reproducible tests. The permutation progress can be read out on block level or block page level. In the case of an error or a bus hang, exactly the same behavior can be repeated for reproduction of the error. Alternatively, the test can be continued after that error.

- **PCI-X protocol behavior permutations within programmable constraints**

  The software makes it possible to specify the values to be varied for each requester-initiator, completer-target, completer-initiator, and requester-target separately. Thus, testing time can be reduced by focusing on cases of interest. Problems can be quickly isolated.

- **Detailed report**

  The software provides a printable report, which shows which protocol behaviors are completely permutated against which other protocol behaviors after how many of data transfers.

- **Predictable testing time**

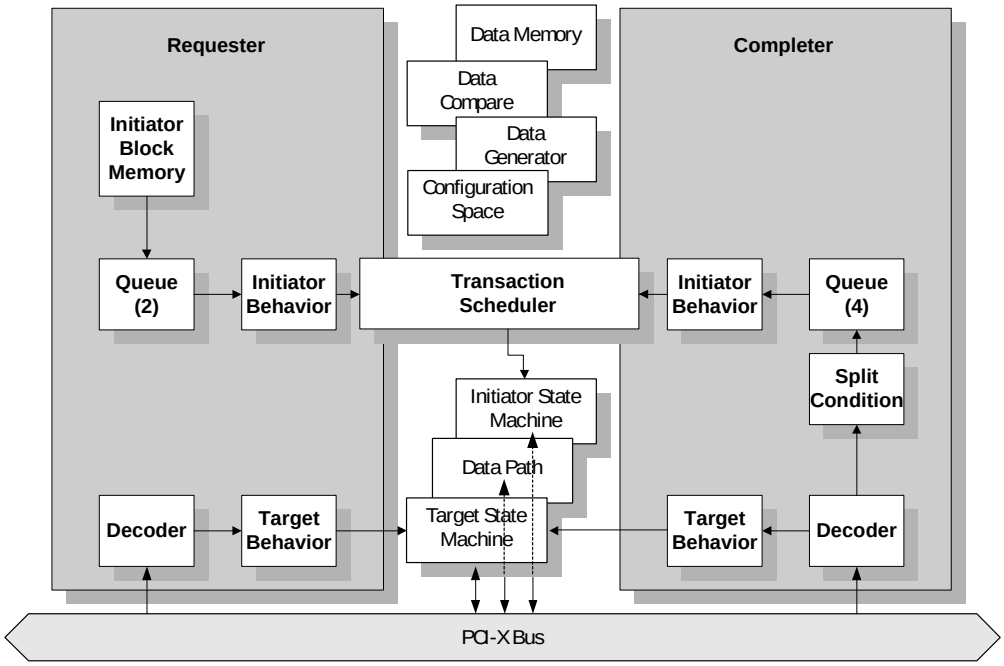
  The test's run time estimated by the PCI-X Protocol Permutation and Randomization can also be written to the report.

Programming the Exerciser

The PCI-X Exerciser testcard can simulate a requester-initiator, a completer-target, a completer-initiator or a requester-target device or all together at the same time. All these can be controlled by functions of the C-API.

This enables the testcard to emulate and/or test any device in your system under test.

The following figure shows the programmable components of the testcard's exerciser.



The concept for programming the exerciser is as follows:

Defining Blocks

1. Defining up to 256 requester-initiator data blocks

   The blocks describe **what** data is transferred over the PCI-X bus. Each block transaction coming out of the **requester-initiator block memory** is put into one of two **queues**.

   See *"Programming Requester-Initiator Block Transfers" on page 30*.

Defining Behaviors

2. Defining **requester-initiator (RI) behaviors**

   The behaviors describe **how** data transferred over the PCI-X bus is executed.

   If any target replies to a transfer and requests a split transaction, the data block attributes are moved internally to the split transaction map for further use. The split transaction map can manage up to 32 split transactions.

   When completing split transactions, the requester-target behaviors are used to control the transfer.

   The transaction that will be given a split response is determined by the split response condition.

   See *"Programming the Behavior of Block Transfers" on page 35*.

3. Defining **completer-target (CT) behaviors**

   The completer-target behaviors control how the target reacts to requests. The completer-target behaviors control, for example, whether a target is able to reply to a transfer with a split response.

   The completer-target can manage up to five transactions.

   See *"Programming the Completer-Target Behavior" on page 48* for more information.

4. Setting the **split response condition**

   To identify a request that will be given a split response, the split response condition property must be set.

   See *"Programming a Split Condition" on page 53* for more information.

5. Defining **completer-initiator (CI) behaviors**

   The completer-initiator behaviors define when and how requests are completed that have been replied to with a split response.

   See *"Programming the Completer-Initiator Behavior" on page 57* for more information.

6. Defining **requester-target (RT) behaviors**

   When completing split transactions, the requester-target behaviors are used to control the transfer.

   See *"Programming the Requester-Target Behavior" on page 63* for more information.

7. Programming the **transaction scheduler**

   The transaction scheduler decides which transactions (completer-initiator or requester-initiator transactions) are performed.

   See *"Scheduling Block Transfers and Split Completions" on page 68* for more information.

8. Programming **completer-target (CT) decoders** for different types of accesses (I/O, memory, configuration cycles, expansion ROM) and the **requester-target decoder** for decoding split completion transactions.

   See *"Programming a Target Decoder" on page 41* for more information.

Resources

9. Defining the data to be transferred

   All data needed for performing transactions is supplied by the onboard **data memory** or from the onboard **real-time generator**.

NOTE The **data path** is shared by initiator and target as a common resource.

   See *"Programming the Data Memory" on page 81* and *"Programming the Data Generator" on page 73* for more information.

10. Performing data compare

    The real-time **data compare** unit is used to compare data that is written to the memory against the actual memory content.

    Data compare can also be performed on the data generator.

11. Accessing the configuration space of the testcard

    The configuration space of the testcard (public and private section) can be employed as a resource. In order to simulate all the possible types of PCI-X devices, the configuration space header of the testcard is freely programmable.

    See *"Programming the Configuration Space" on page 45* for more information.

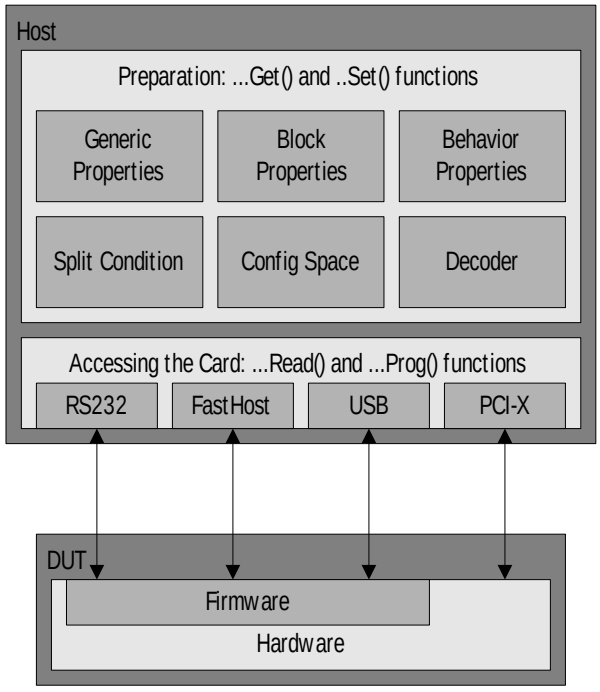
The exerciser further allows you to generate PCI-X interrupts. See *"Programming PCI-X Interrupts" on page 85*.

Reading From and Writing To the Memories

The C-API uses preparation functions (…`Get()` and …`Set()` functions) for reading and writing properties from and to the host. These functions do not access the hardware.

To access the hardware, use the …Read(), …Prog(), or …Write() functions.

The following figure shows the C-API architecture.



Downloading Settings and Running the Exerciser

This section describes global exerciser functions.

Downloading

All available properties first are programmed to the host storage. If you set all properties, you have to write them to the testcard.

To write all settings to the testcard, use *BestXExerciserProg*.

Reading

To read settings from the testcard, use *BestXExerciserRead*. This function can either read the entire memory or only generic properties.

Running the Exerciser

To run the exerciser, use *BestXExerciserRun*.

Stopping the Exerciser

To stop the exerciser, use *BestXExerciserStop*, where the current transaction will be completed.

Exerciser Reset

To reset all bus state machines (initiator and target) and to clear the requester-initiator intention, use *BestXExerciserReset*.

CAUTION

The call *BestXExerciserReset* might lead to a behavior that does not conform to the protocol. All state machines are reset, regardless of their current state. Hence, the bus state will be unclear. This call should only be used if a bus is hung.

Programming the Exerciser as a Requester-Initiator Device

To program the testcard's exerciser as a requester-initiator device means programming the testcard to initiate data transfers via PCI-X bus either to a target device under test, or to the testcard's own target (completer-target). The latter test case can be used to increase bus load.

The following need to be programmed for the testcard to perform data transfer:

1. Generic requester-initiator properties

   Generic requester-initiator properties determine the behavior of the testcard and are valid during a complete exerciser run. They determine, for example, whether the requester-initiator should start immediately or conditionally after a trigger event.

   See *"Programming Generic Requester-Initiator Properties" on page 29*.

2. The requester-initiator transactions to be performed

   Transactions can be summarized into blocks. The properties of each block and its transactions, such as PCI-X bus address, number of dwords to be transferred or the bus command, are programmed in the block transfer memory.

   See *"Programming Requester-Initiator Block Transfers" on page 30*.

3. The behaviors to be used with the transactions

   The behaviors determine how the transactions should be executed, for example, whether and how often the requester-initiator disconnects its current sequence. This information is located in the requester-initiator behavior memory.

   See *"Programming the Behavior of Block Transfers" on page 35*.

4. The data to be used for the transactions

   For this purpose, the data memory and the data generator can be used as a data resource by the requester-initiator.

   See *"Programming the Data Memory" on page 81* and *"Programming the Data Generator" on page 73*.

5. The way of running the exerciser

   See *"Downloading Settings and Running the Exerciser" on page 27*.

Programming Generic Requester-Initiator Properties

Generic requester-initiator properties are valid for a complete exerciser run.

You can program the following generic requester-initiator properties:

- How many block transfers were programmed (`BX_RIGEN_NUMBLK`)

  Valid values are `1 … 256`.

- How often the block transfers will be repeated (`BX_RIGEN_REPEATBLK`)

  Valid values are `1 … 0xffffffff, BX_RIGEN_REPEATBLK_INFINITE`

- The number of valid RI-behaviors (`BX_RIGEN_NUMBEH`)

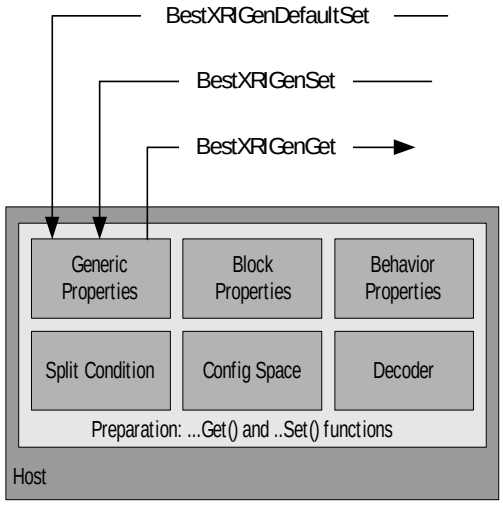
  Valid values are `1 … 256`.

- The values for the seven skip registers (`BX_RIGEN_SKIP_REG1 … BX_RIGEN_SKIP_REG7`)

  Valid values are `0 … 1023`.

How to Program Generic Requester-Initiator Properties

The following figure shows the functions used to program the generic requester-initiator properties.



Programming Steps

To set generic requester-initiator property values on the host:

1. Set all generic requester-initiator properties to default values with *BestXRIGenDefaultSet*.
2. Set each property with *BestXRIGenSet* to the appropriate value.

   To get the value of one property, use *BestXRIGenGet*.

Example for Programming Generic Requester-Initiator Properties

Task Define that two requester-initiator behaviors should be executed twice.

Implementation

```
/* Define that two programmed block transfers are repeated twice */
BX_TRY(BestXRIGenSet(handle, BX_RIGEN_NUMBLK, 2));
BX_TRY(BestXRIGenSet(handle, BX_RIGEN_REPEATBLK, 2));
/* Define that five behaviors for the block transfers are
programmed */
BX_TRY(BestXRIGenSet(handle, BX_RIGEN_NUMBEH, 5));
```

Programming Requester-Initiator Block Transfers

A requester-initiator block transfer means to transfer a contiguous block of data from one place to the other. A block transfer is specified by:

- Read/Write operation

Operation	Source	Destination
Read	physical bus address	exerciser data
Write	exerciser data	physical bus address

- The bus command:
  - Memory read DWORD or memory write
  - I/O read or write, configuration read or write
  - Memory read block or memory write block
  - Split completion
  - Interrupt acknowledge
  - Alias to memory read or write block
  - Reserved cycle
- Size (number of bytes (up to 4 GB))

The information for a block transfer is stored in the requester-initiator block memory. This memory can hold up to 256 block transfer entries.

How to Program Block Transfers

To program requester-initiator block transfers:

1 Set all entries of the requester-initiator block memory to default values.

Use *BestXRIBlockDefaultSet*.

2 Program each block transfer to one line of the requester-initiator block transfer memory. Each transfer is specified by several block transfer properties.

For each block transfer property to be programmed, use *BestXRIBlockSet*.

To query the value of a requester-initiator block property, use *BestXRIBlockGet*.

3 Set all generic requester-initiator properties to default values.

Use *BestXRIGenDefaultSet*.

4 Define how many times each block transfer should be executed.

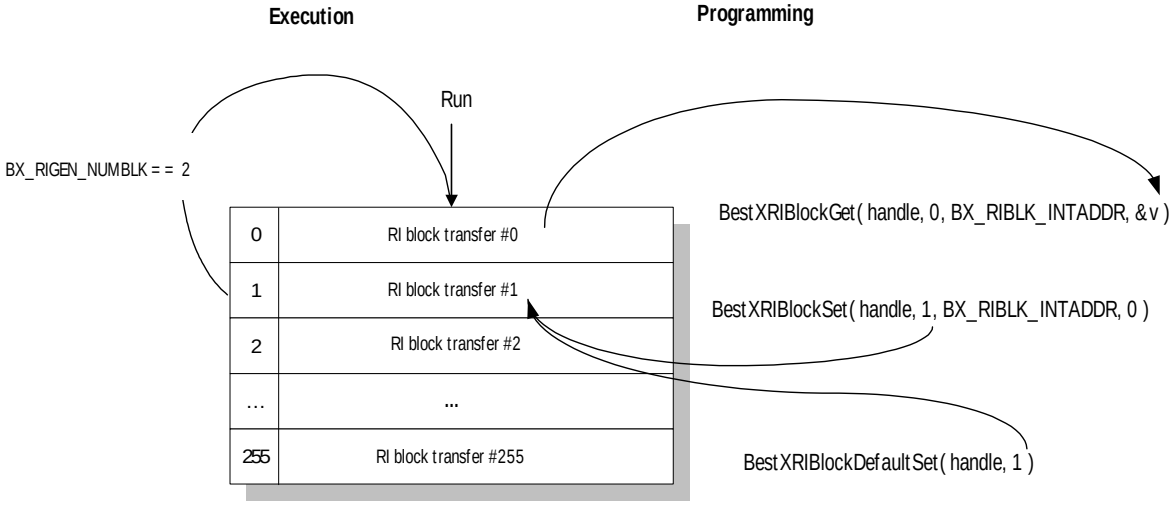
- First, set all generic requester-initiator properties to default values with *BestXRIGenDefaultSet*
- Then, use *BestXRIGenSet* to set the requester-initiator generic property `BX_RIGEN_NUMBLK` to the appropriate value (`1 … 256`).

For more information about generic requester-initiator properties, see *"Programming Generic Requester-Initiator Properties" on page 29*.

5 Define how often the previously specified block transfers should be executed.

Use *BestXRIGenSet* and set the requester-initiator generic property `BX_RIGEN_REPEATBLK` to the appropriate value. Valid values are `BX_RIGEN_REPEATBLK_INFINITE`, `1 … 0xffffffff`.

The following figure shows the memory design, the available functions to program the requester-initiator block memory, and the execution order.



6 Download all exerciser settings and properties to the hardware with *BestXExerciserProg*.

7 Start the transfer(s) with *BestXExerciserRun*.

Examples for Programming Block Transfers

Task (Example 1) The task is as follows:

- Program the E2929A so that it starts two block transfers:
  - First block transfer: One memory read block of 495 bytes from bus address 100030fd\h and
  - Second block transfer: One I/O write of 7 bytes to bus address 1000ff\h.

  Use the internal address starting at 0\h.

Implementation (Example 1)

```
/* Set the number of block transfers to be executed and how
often they should be executed */

BX_TRY(BestXRIGenDefaultSet(handle);
BX_TRY(BestXRIGenSet(handle, BX_RIGEN_NUMBLK, 2));
BX_TRY(BestXRIGenSet(handle, BX_RIGEN_REPEATBLK, 1));

/* Program a block transfer to memory line 0*/

BX_TRY(BestXRIBlockDefaultSet(handle, 0));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_BUSADDR_LO,
                       0x100030fdUL));

BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_BUSCMD,
                       BX_RIBLK_BUSCMD_MEM_READBLOCK));

BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_NUMBYTES, 495));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_INTADDR, 0));

/* Program a block transfer to memory line 1 */

BX_TRY(BestXRIBlockDefaultSet(handle, 1));

BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_BUSADDR_LO,
                       0x1000ffUL));

BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_BUSCMD,
                       BX_RIBLK_BUSCMD_IO_WRITE));

BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_NUMBYTES, 7));
BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_INTADDR, 0));

/* Write the settings to the testcard and run the exerciser */

BX_TRY(BestXExerciserProg(handle));
BX_TRY(BestXExerciserRun(handle));
```

Task (Example 2)

The task is as follows:

- Program the same block transfers as in example 1.

  This time:

  – Use the data generator as resource.

  – Put the memory read with 531 bytes into queue B and the I/O write into queue A.

  – Set the *relaxorder* and *nosnoop* bits.

  – Let the exerciser repeat the transfers 5 times.

Implementation (Example 2)

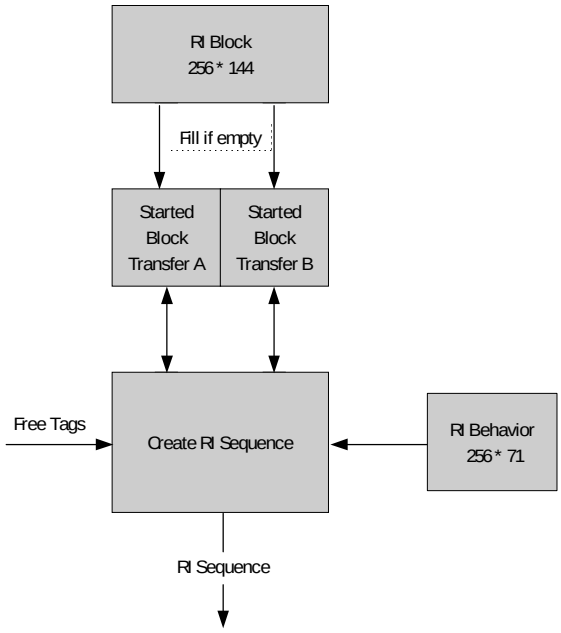
```
/* Define that two block transfers are programmed that are repeated
5 times */

BX_TRY(BestXRIGenSet(handle, BX_RIGEN_NUMBLK, 2));
BX_TRY(BestXRIGenSet(handle, BX_RIGEN_REPEATBLK, 5));

/* Program a block transfer to memory line 0*/

BX_TRY(BestXRIBlockDefaultSet(handle, 0));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_BUSADDR_LO,
                       0x100030fdUL));

BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_BUSCMD,
                       BX_RIBLK_BUSCMD_MEM_READBLOCK));

BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_NUMBYTES, 531));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_INTADDR, 0));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_RELAXORDER, 1));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_NOSNOOP, 1));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_RESOURCE,
                       BX_RIBLK_RESOURCE_DATAGEN));

BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_QUEUE,
                       BX_RIBLK_QUEUE_B));

/* Program a block transfer to memory line 1 */

BX_TRY(BestXRIBlockDefaultSet(handle, 1));

BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_BUSADDR_LO,
                       0x1000ffUL));

BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_BUSCMD,
      BX_RIBLK_BUSCMD_IO_WRITE));

BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_NUMBYTES, 7));
BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_INTADDR, 0));
BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_RELAXORDER, 1));
BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_NOSNOOP, 1));

BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_RESOURCE,
                       BX_RIBLK_RESOURCE_DATAGEN));

BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_QUEUE,
                       BX_RIBLK_QUEUE_A));

/* Write the settings to the testcard and run the exerciser */

BX_TRY(BestXExerciserProg(handle));
BX_TRY(BestXExerciserRun(handle));
```

Programming the Behavior of Block Transfers

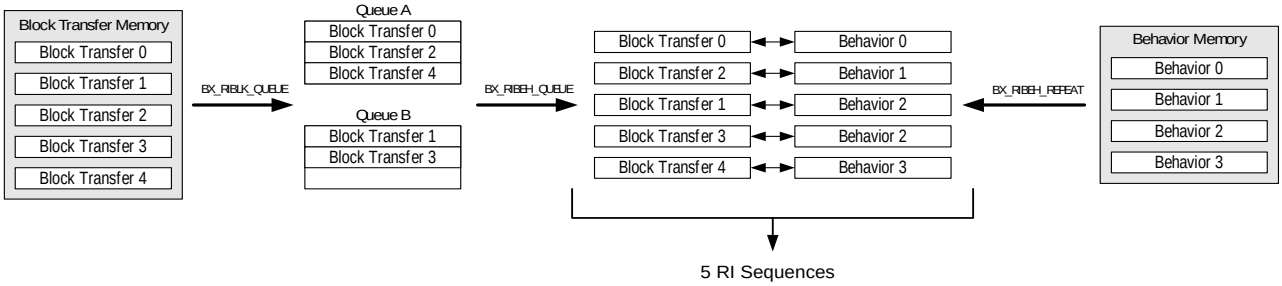
The requester-initiator behaviors define how block transfers are executed. In particular, behaviors control the partitioning of blocks into sequences and the reordering of blocks.

The requester-initiator behaviors are stored in the requester-initiator behavior memory. Each memory line holds one behavior for the block transfer. When the exerciser is started, one behavior is assigned to one block transfer.

The following figure shows the correlation between block transfers and behaviors, and how to generate sequences.



How to create sequences in detail is shown in the following figure by means of an example.



Each requester-initiator behavior has the following programmable properties.

- Requester-initiator queue (`BX_RIBEH_QUEUE`)

  Valid values are `BX_RIBEH_QUEUE_A`, `BX_RIBEH_QUEUE_B`, `BX_RIBEH_QUEUE_NEXT`.

  By setting this behavior and the corresponding block property (`BX_RIBLK_QUEUE`), you can determine if blocks get executed in order or if they bypass each other.

- Fixed tag number or any free tag if possible (`BX_RIBEH_TAG`)

- Byte count for the sequence (`BX_RIBEH_BYTECOUNT`)

  Valid values are 1 … 4096.

  This property partitions the value of the block transfer property `BX_RIBLK_NUMBYTES` into sequences of a maximum length of 4096 bytes.

- Disconnect at ADB number N (`BX_RIBEH_DISCONNECT`)

  This property allows you to break a sequence into multiple transactions. The requester-initiator disconnects at every N-th allowable disconnect boundary (ADB). Typically, the requester-initiator will not disconnect, because the data transfer will have been completed when it requests a transaction.

  When the requester-initiator disconnects a sequence, it resumes the disconnected sequence. It is not possible to execute some other action in between.

- Clock delay before assertion of `REQ#` (`BX_RIBEH_DELAY`)

  This property allows you to vary latencies between transactions. Sometimes the minimum achievable latency to the next requester-initiator transaction is restricted by the most recent event and sometimes by the data path configuration.

- Number of address steps (`BX_RIBEH_STEPS`)

  The number of address steps is the number of clock cycles between the assertion of `GNT#` and the assertion of `FRAME#` plus two clock cycles. These two clock cycles are designed into the register-to-register interface of PCI-X.

- 64-bit data transfer request (`BX_RIBEH_REQ64`)

  Valid values are:

  – 1 = Assert the REQ64# signal

  – 0 = don't assert the REQ64# signal

- Number of repeats (`BX_RIBEH_REPEAT`)

  The current behavior is repeated N times before the next behavior is used. Valid values are 1 … 256.

- Tag number to be used for this sequence (`BX_RIBEH_TAG`)

  Valid values are 0 … 31.

- Number of clock cycles that REQ# is asserted after the address phase (`BX_RIBEH_RELREQ`)

  Valid values are 1 … 2047.

- Number of the skip register (`BX_RIBEH_SKIP`)

  Valid values are 1 … 7, `BX_RIBEH_SKIP_NO`.

  The number is used if this behavior was repeated. The value of the selected skip register is then added to the start address of the next sequence.

How to Program the Behavior of Block Transfers

To program the behavior of requester-initiator block transfers:

1 Set all entries of the requester-initiator behavior memory to default values.

Use *BestXRIBehDefaultSet*.

2 Program each behavior to one line of the requester-initiator behavior memory. Each behavior is specified by several behavior properties.

For each behavior property to be programmed, use *BestXRIBehSet*.

To query the value of a requester-initiator behavior property, use *BestXRIBehGet*.

3 Define how many behaviors should be executed.

– First, set all generic requester-initiator properties to default values with *BestXRIGenDefaultSet*.

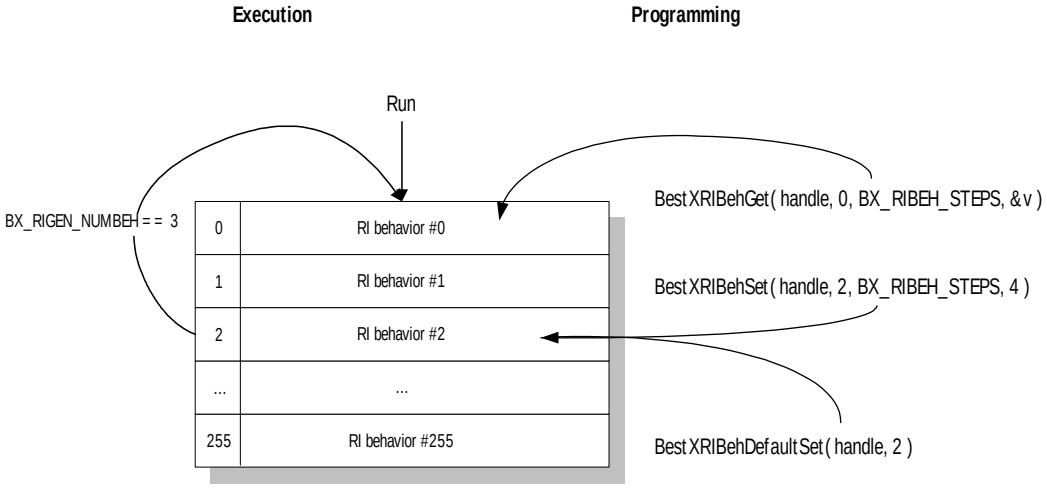
– Use *BestXRIGenSet* and set the requester-initiator generic property `BX_RIGEN_NUMBEH` to the appropriate value (`1 … 256`).

For more information about generic requester-initiator properties, see *"Programming Generic Requester-Initiator Properties" on page 29*.

4 Define how often the current behavior is applied before the next behavior is used.

Use *BestXRIBehSet* and set the requester-initiator generic property `BX_RIBEH_REPEAT` to the appropriate value. Valid values are `1 … 256`.

The following figure shows the memory design, the available functions to program the requester-initiator block memory, and the execution order.



5 Download all exerciser settings and properties to the hardware with *BestXExerciserProg*.

6 Start the transfer(s) with *BestXExerciserRun*.

Example for Programming the Behavior of Block Transfers

Task Perform the following task:

- Program the same block transfers as in *"Task (Example 2)" on page 33*.
- Let the requester-initiator disconnect the *ReadBlock* at each ADB. The maximum sequence length for the *ReadBlock* transfers should be 321 bytes.
- Use sequence tag #30 for the I/O write transfer.
- Apply the I/O and ReadBlock addresses 3 clocks before asserting FRAME#.

Implementation

```
/* Program the behavior of the block transfers */

BX_TRY(BestXRIGenDefaultSet(handle);
BX_TRY(BestXRIGenSet(handle, BX_RIGEN_NUMBEH, 5));

/* Program the behavior properties steps, bytecount, and disconnect
to behavior memory line 0 */

BX_TRY(BestXRIBehDefaultSet(handle, 0));
BX_TRY(BestXRIBehSet(handle, 0, BX_RIBEH_STEPS, 3));
BX_TRY(BestXRIBehSet(handle, 0, BX_RIBEH_BYTECOUNT, 321));
BX_TRY(BestXRIBehSet(handle, 0, BX_RIBEH_DISCONNECT, 1));

/* Program the behavior properties steps, bytecount, and disconnect
to behavior memory line 1 */

BX_TRY(BestXRIBehDefaultSet(handle, 1));
BX_TRY(BestXRIBehSet(handle, 1, BX_RIBEH_STEPS, 3));
BX_TRY(BestXRIBehSet(handle, 1, BX_RIBEH_BYTECOUNT, 321));
BX_TRY(BestXRIBehSet(handle, 1, BX_RIBEH_DISCONNECT, 1));

/* Program the behavior properties steps and the tag to behavior
memory line 2 */

BX_TRY(BestXRIBehDefaultSet(handle, 2));
BX_TRY(BestXRIBehSet(handle, 2, BX_RIBEH_STEPS, 3));
BX_TRY(BestXRIBehSet(handle, 2, BX_RIBEH_TAG, 30));
```

```
/* Program the behavior properties steps and the tag to behavior
memory line 3 */

BX_TRY(BestXRIBehDefaultSet(handle, 3));
BX_TRY(BestXRIBehSet(handle, 3, BX_RIBEH_STEPS, 3));
BX_TRY(BestXRIBehSet(handle, 3, BX_RIBEH_TAG, 30));

/* Program the behavior properties steps and the tag to behavior
memory line 4 */

BX_TRY(BestXRIBehDefaultSet(handle, 4));
BX_TRY(BestXRIBehSet(handle, 4, BX_RIBEH_STEPS, 3));
BX_TRY(BestXRIBehSet(handle, 4, BX_RIBEH_TAG, 30));

/* Program the behaviors to the testcard's exerciser and run the
exerciser. */

BX_TRY(BestXExerciserProg(handle));
BX_TRY(BestXExerciserRun(handle));
```

Programming the Exerciser as a Completer-Target Device

A PCI-X completer-target device is the target of a transaction. It decodes all PCI-X bus transactions except split completion transactions.

To program the exerciser so that it acts as a PCI-X completer-target device, you have to:

- Set up the target decoders provided by the testcard for I/O, memory and configuration cycles.

  See *"Programming a Target Decoder" on page 41*.

  Because the decoders are closely linked to entries in the configuration space, you can program the decoders' base address, size and prefetch also by modifying the configuration space header. See *"Programming the Configuration Space" on page 45*.

NOTE The option to completely program the configuration space header is provided for very sophisticated tests, for example, to test systems without a BIOS or with a very rudimentary BIOS.

- Define how the completer-target reacts to transactions driven onto the bus.

  See *"Programming the Completer-Target Behavior" on page 48*.

- Define how many behaviors are used before the first one is used again.

  See *"Programming Generic Completer-Target Properties" on page 52*.

- Specify the internal data resources.

  See *"Programming the Data Memory" on page 81* and *"Programming the Data Generator" on page 73*.

Programming a Target Decoder

The completer-target needs the address range information to decide whether it has to react to transactions driven onto the bus. There are different types of decoders for different types of accesses (I/O, memory, configuration cycles).

Types of Decoders

The testcard provides the following target decoders:

- Three programmable memory decoders (six bars) for memory or I/O accesses (`BX_DEC_BAR0 … BX_DEC_BAR5`)

  These decoders are programmable regarding size, location, prefetch, compare and resource.

- Expansion ROM Decoder (`BX_DEC_EXPROM`)

  This decoder is programmable regarding size and resource (flash).

- Configuration Decoder (`BX_DEC_CONFIG`)

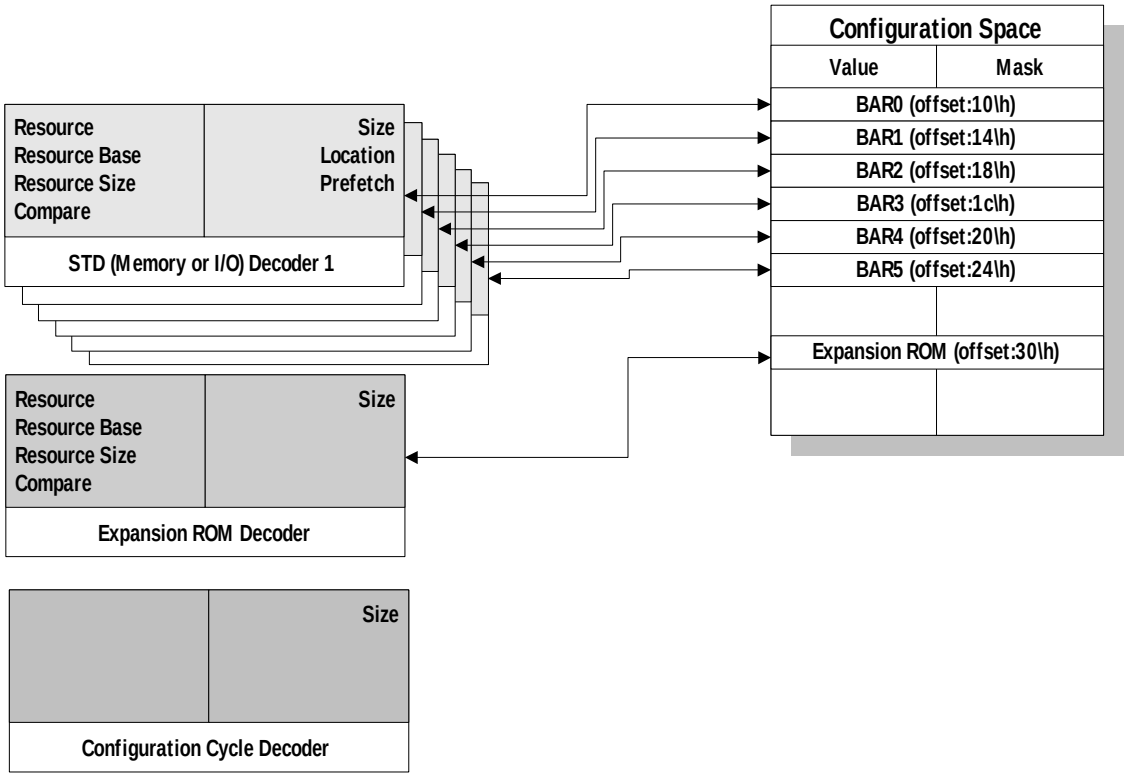
  This decoder is only programmable regarding size (on/off) and reacts on PCI-X and PCI accesses.

NOTE The decoders are closely linked to entries in the configuration space. See the following figure.

Decoders Linked to the Configuration Space Header

The following figure shows the different decoders, the respective entries in the configuration space header and how both can be programmed.

For programming the configuration space header, see *"Programming the Configuration Space" on page 45*.



Standard and expansion ROM decoders are equipped with a set of parameters that define their properties, such as whether the decoded memory space is prefetchable, or which data resource they are connected to. The configuration cycle decoder is only programmable regarding its size.

How to Program a Decoder

To program a decoder:

1 Set all target decoder properties on the host to default values. Use *BestXTDecoderDefaultSet*.

2 To set the properties for the target decoder on the host, you can:

- Set all properties for a BAR decoder (BAR 0 … 5) with *BestXTDecoderAllSet*.
- Set one property for one decoder (BAR 0 … 5, EXPROM and configuration decoder) with *BestXTDecoderSet*.

  To get a property value, use *BestXTDecoderGet*.

Dependencies between decoder and their programmable properties are shown in the following tables.

Programmable Properties

Which properties can be programmed for which decoder is shown in the following table:

Decoder(s)	Property	Value(s)
Standard EXPROM CONFIG	BX_DECP_SIZE	Memory: 0, 2 … 63 I/O: 0, 2 … 31 ExpROM: 0, 11 … 24 Config: 0, 1
Standard	BX_DECP_LOCATION	BX_DECP_LOCATION_MEM BX_DECP_LOCATION_IO
Standard	BX_DECP_PREFETCH	0, 1
Standard EXPROM	BX_DECP_RESOURCE	BX_DECP_RESOURCE_MEM BX_DECP_RESOURCE_GEN BX_DECP_RESOURCE_FLASH
Standard EXPROM	BX_DECP_RESBASE	0 … 224 in steps of 4
Standard EXPROM	BX_DECP_RESSIZE	2… 24
Standard	BX_DECP_COMPARE	0, 1

Depending on the selected resource (`BX_DECP_RESOURCE`), programming the behavior of the completer-target is restricted. See the following table.

Programmable Behaviors

The following table shows the dependencies between resource and behavior.

Resource	Decoder(s)	Behavior
Memory, Generator	Standard EXPROM	Programmable
Flash	EXPROM	Fixed, retries until data available, single data phase disconnect
Config space	CONFIG	Fixed

3 Write the properties to the testcard with *BestXExerciserProg*.

Example for Programming a Decoder

Task Set up a standard decoder with the following properties:

- Decoder size 12, Memory, Prefetch
- Resource: Date memory, resource size 12, internal resource base address 0
- Base Address: 0x10003000

Implementation **/* Setup the memory decoder (baseaddress = 10003000\h, size=c\h) */**

```
bx_int32 dsize=0xcUL;
bx_int32 bbase=0x10003000UL;

BX_TRY(BestXTDecoderDefaultSet(handle));
BX_TRY(BestXTDecoderSet(handle, BX_DEC_BAR0, BX_DECP_LOCATION,
                        BX_DECP_LOCATION_MEM));

BX_TRY(BestXTDecoderSet(handle, BX_DEC_BAR0, BX_DECP_SIZE, dsize));
BX_TRY(BestXTDecoderSet(handle, BX_DEC_BAR0, BX_DECP_PREFETCH, 1));

BX_TRY(BestXTDecoderSet(handle, BX_DEC_BAR0, BX_DECP_RESOURCE,
                        BX_DECP_RESOURCE_MEM));

BX_TRY(BestXTDecoderSet(handle, BX_DEC_BAR0, BX_DECP_RESBASE,
                        0x0));

BX_TRY(BestXTDecoderSet(handle, BX_DEC_BAR0, BX_DECP_RESSIZE,
                        dsize));
```

Programming the Configuration Space

You can program every single register of the testcard's configuration space header.

Usually, all settings in the configuration space header that need to be made when starting the system are either made by the BIOS or by programming target decoders. The latter applies to settings concerning the base address registers.

NOTE The option to completely program the configuration space header is provided for very sophisticated tests, for example, to test systems without a BIOS or with a very rudimentary BIOS.

For every bit of the various registers in the configuration space header, you can determine whether it is fixed (read-only) or programmable from outside (BIOS) (read/writeable). For both types, values can be specified as required for BIOS configuration during system startup:

- Determine **read-only** values, for example, a "Vendor ID", which can then be evaluated by the BIOS.
- Determine **read/writeable** values, for example, base address register entries. They can be used by the BIOS to determine the wanted size of the decoded address range and will then be overwritten with the actual base address.

How to Modify the Configuration Space

To modify the configuration space header:

1 Set the register value and the register mask that configures the status of a bit within a register on the host.

   – To set a register mask, use *BestXConfRegMaskSet*.

   In the 32-bit mask:

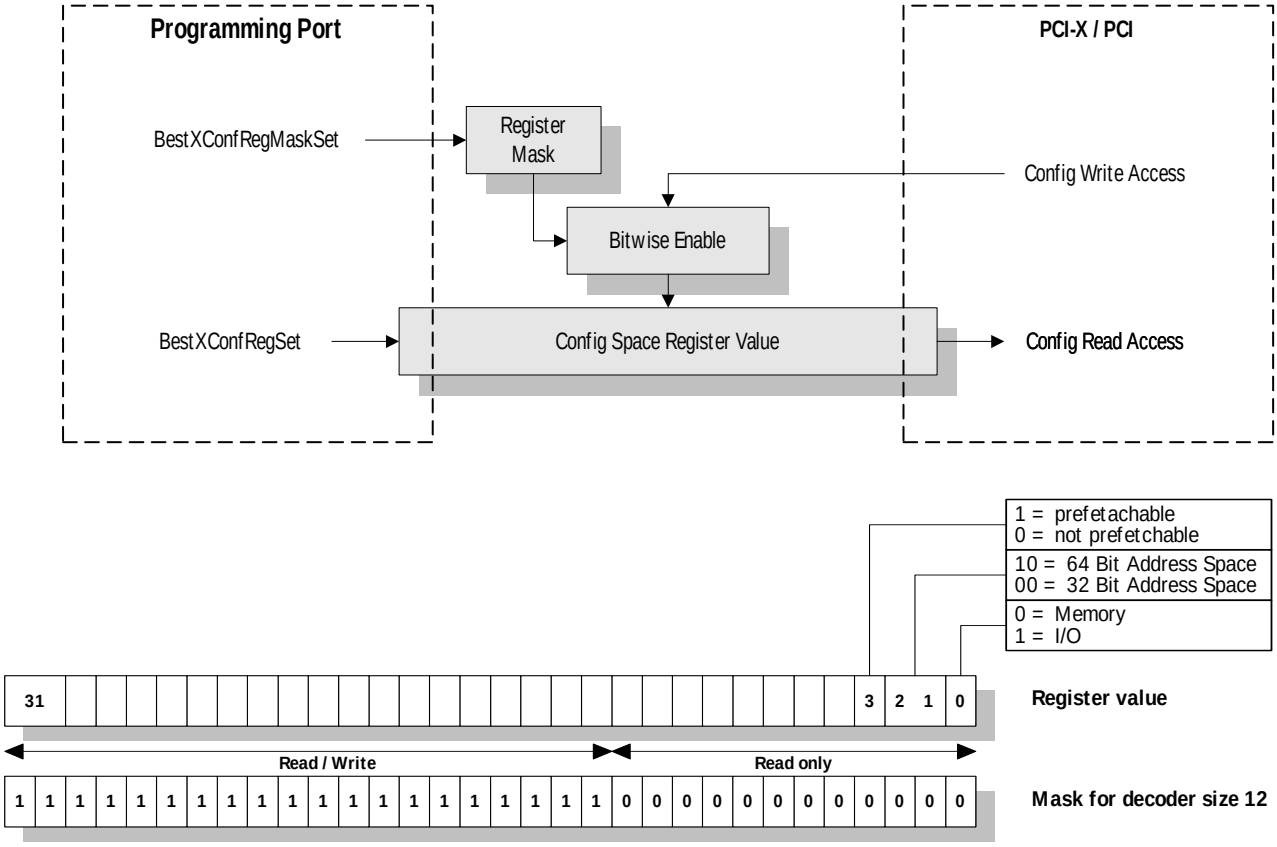
   0 means: This bit is fixed (read-only) and its value cannot be changed.

   1 means: This bit is programmable.

   – To set the value of a register, use *BestXConfRegSet*.

CAUTION Do not set several address spaces to the same decoding location, because the system under test can crash or could even be damaged.

The following figure shows the dependencies for accessing the configuration space between the programming port (C-API) and the PCI-X/PCI interface (BIOS).



2 Write the values to the testcard with *BestXExerciserProg*.

Example for Modifying the Configuration Space

Task Set up a I/O decoder with base address = 100000\h and size=a\h and define the data generator as data source for the data transfer.

Implementation

```
// Set up the I/O decoder (base address = 100000\h, size=a\h
dsize=0xaUL;
bbase=0x100000UL;

/* Specify the Standard decoder BAR 4 to claim transactions to I/O
address ranges. This automatically sets the size to default. */
BX_TRY(BestXTDecoderSet(handle, BX_DEC_BAR4, BX_DECP_LOCATION,
                        BX_DECP_LOCATION_IO));
/* Program the mask. Offset for BAR 4 is 20/h. See also the figure
in "Decoders Linked to the Configuration Space Header" on page 68.*/
BX_TRY(BestXConfRegMaskSet(handle, 0x20UL, ((~0)<<dsize)));
/* Specify the base address and location */
BX_TRY(BestXConfRegSet(handle, 0x20UL, (bbase & ((~0)<<dsize))+1));
/* Specify the data generator as source for the data transfer and
define the base address and the size for the source */
BX_TRY(BestXTDecoderSet(handle, BX_DEC_BAR4, BX_DECP_RESOURCE,
                        BX_DECP_RESOURCE_GEN));
BX_TRY(BestXTDecoderSet(handle, BX_DEC_BAR4, BX_DECP_RESBASE,
                        0x40000));
BX_TRY(BestXTDecoderSet(handle, BX_DEC_BAR4, BX_DECP_RESSIZE,
                        dsize));
```

Programming the Completer-Target Behavior

After setting up and enabling the decoders and base address registers, the Agilent PCI-X Exerciser is able to react to accesses from requester-initiator devices. The behavior of how it reacts can be programmed by setting completer-target behavior properties.

The completer-target behaviors are stored in the completer-target behavior memory. Each memory entry holds one behavior; these are successively used for each request.

Each completer-target behavior has the following programmable properties:

- Decode speed A, B or C (`BX_CTBEH_DECSPEED`)

NOTE Decode speed A is only supported up to 66 MHz.

- Accept 64-bit wide data transfer (`BX_CTBEH_ACK64`)

  With this attribute, the target signals that it is capable of accepting 64-bit accesses.

  – 1 = Assert the ACK64# signal

  – 0 = don't assert ACK64#

- Initial target response (`BX_CTBEH_INITIAL`)

  This property defines how the completer-target responds after waiting the number of cycles specified with `BX_CTBEH_LATENCY`.

  If the split condition is met, this behavior is ignored and a split response is given after the number of wait cycles defined with `BX_CTBEH_SPLITLATENCY`.

  For more information about split response conditions, see *"Programming a Split Condition" on page 53*.

- Number of initial wait states (`BX_CTBEH_LATENCY`)

  Valid values are `3 … 34`.

- Subsequent target response (`BX_CTBEH_SUBSEQ`)

  This property specifies the target response in subsequent data phases. It comes only into effect if the corresponding `BX_CTBEH_INITIAL` completer-target behavior was set to `BX_CTBEH_INITIAL_ACCEPT`.

- Subsequent data phases (`BX_CTBEH_SUBSEQPHASE`)

  Number of clock cycles for `BX_CTBEH_SUBSEQ`. Valid values for the subsequent data phases are `0 … 2047`.

- Split latency (`BX_CTBEH_SPLITLATENCY`)

  Number of wait states until a split response is signaled. Only valid if a split condition is true.

- Split response enable (`BX_CTBEH_SPLITENABLE`)

  This property defines if a split response is generated. You must set up the decoder in the Target Decode window accordingly for this property to have an effect.

- Number of repeats (`BX_CTBEH_REPEAT`)

  The current behavior is repeated N times before the next behavior is used. Valid values are `1 … 65536`.

How to Program the Completer-Target Behavior

To program the completer-target behavior:

1. Set all entries of the completer-target behavior memory to default values.

   Use *BestXCTBehDefaultSet*.

2. Program each behavior to one line of the completer-target behavior memory. Each behavior is specified by several behavior properties.

   For each behavior property to be programmed, use *BestXCTBehSet*.

   To query the value of a completer-target behavior property, use *BestXCTBehGet*.

3. Set all generic completer-target properties to default values.

   Use *BestXCTGenDefaultSet*.

4. Define how many behaviors should be executed.

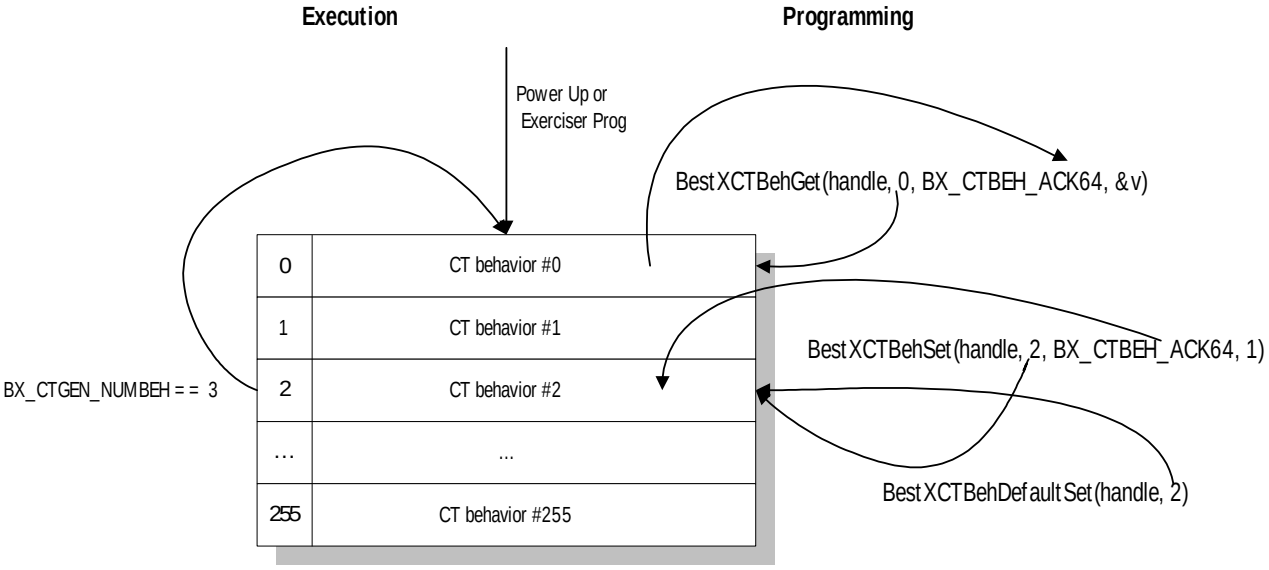
   – Use *BestXCTGenSet* and set the completer-target generic property `BX_CTGEN_NUMBEH` to the appropriate value (`1 … 256`).

   For more information about generic completer-target properties, see *"Programming Generic Completer-Target Properties" on page 52*.

5. Define how often the current behavior is applied before the next behavior is used.

   Use *BestXCTBehSet* and set the completer-target generic property `BX_CTBEH_REPEAT` to the appropriate value. Valid values are `1 … 65536`.

The following figure shows the memory design, the available functions to program the completer-target behavior memory, and the execution order.



6 Download all exerciser settings and properties to the hardware with *BestXExerciserProg*.

Example for Programming the Completer-Target Behavior

Task Perform the following task:

- Set up the completer-target so that it:
  - Uses decode speed A, B and C
  - Signal RETRY 3 times
- Program single data phase disconnect and disconnect at next ADB.
- Initial latency should be 8.
- Signal split response on an access to address range `0x100032XX` with memory read block.

Implementation

/* Program that 3 completer-target behaviors are used before the first one is used again */

```
BX_TRY(BestXCTGenDefaultSet(handle));

BX_TRY(BestXCTGenSet(handle, BX_CTGEN_NUMBEH, 3));

/* Program the behavior properties decode speed, initial, repeat
and latency to behavior memory line 0 */

BX_TRY(BestXCTBehDefaultSet(handle, 0));

BX_TRY(BestXCTBehSet(handle, 0, BX_CTBEH_DECSPEED,
                     BX_CTBEH_DECSPEED_A));

BX_TRY(BestXCTBehSet(handle, 0, BX_CTBEH_INITIAL,
                     BX_CTBEH_INITIAL_RETRY));

BX_TRY(BestXCTBehSet(handle, 0, BX_CTBEH_REPEAT, 3));

BX_TRY(BestXCTBehSet(handle, 0, BX_CTBEH_LATENCY, 8));

/* Program the behavior properties decode speed and initial to
behavior memory line 1 */

BX_TRY(BestXCTBehDefaultSet(handle, 1));

BX_TRY(BestXCTBehSet(handle, 1, BX_CTBEH_DECSPEED,
                       BX_CTBEH_DECSPEED_B));

BX_TRY(BestXCTBehSet(handle, 1, BX_CTBEH_INITIAL,
                       BX_CTBEH_INITIAL_SINGLE));

/* Program the behavior properties decode speed, initial, and the
target response in subsequent data phases to behavior memory line 2
*/

BX_TRY(BestXCTBehDefaultSet(handle, 2));

BX_TRY(BestXCTBehSet(handle, 2, BX_CTBEH_DECSPEED,
                       BX_CTBEH_DECSPEED_C));

BX_TRY(BestXCTBehSet(handle, 2, BX_CTBEH_INITIAL,
                       BX_CTBEH_INITIAL_ACCEPT));

BX_TRY(BestXCTBehSet(handle, 2, BX_CTBEH_SUBSEQ,
                       BX_CTBEH_SUBSEQ_DISCONNECT));

BX_TRY(BestXCTBehSet(handle, 2, BX_CTBEH_SUBSEQPHASE, 1));
```

Programming Generic Completer-Target Properties

Generic completer-target properties are valid for a complete exerciser run.

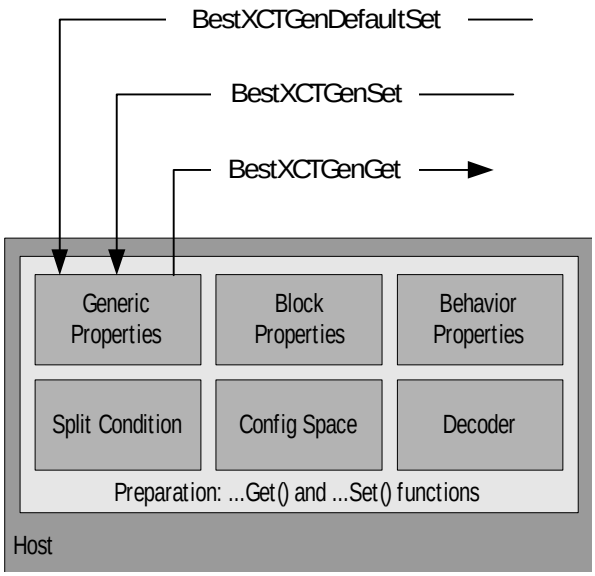
You can program the following generic requester-initiator property:

- How many completer-target behaviors are used before the first one is used again (`BX_CTGEN_NUMBEH`)

  Valid values are `1 … 256`.

How to Program Generic Completer-Target Properties

The following figure shows the functions used to program the generic completer-target properties.



Programming Steps

To set generic completer-target property values on the host:

1. Set all generic requester-initiator properties memory to default values with *BestXCTGenDefaultSet*.
2. Set each property with *BestXCTGenSet* to the appropriate value.

   To get the value of one property, use *BestXCTGenGet*.

Example for Programming Generic Completer-Target Properties

Task Program that three behaviors (0 … 2) are executed before the first one is used again.

Implementation

```
/* Define that three behaviors are used before the first one is
used again */

BX_TRY(BestXCTGenDefaultSet(handle));
BX_TRY(BestXCTGenSet(handle, BX_CTGEN_NUMBEH, 3));
```

Programming a Split Condition

Split condition properties identify a particular type of requests that shall be given a split response and determine in which request queue those requests will be put for later completion. You can define up to four different request types (split decoder 0 … 3).

The request that shall be given a split response can be identified by:

- The address value shown in the address phase

  This is programmed by setting the address value (`BX_CTSPLIT_ADDRVAL_HI` and `BX_CTSPLIT_ADDRVAL_LO`) and a mask (`BX_CTSPLIT_ADDRMASK_HI` and `BX_CTSPLIT_ADDRMASK_LO`).

  The mask defines which bits of the AD signal must be equal to the corresponding bit in the address value:

  – Bit = 0: The corresponding AD[n] signal is don't care.

  – Bit = 1: The corresponding AD[n] signal must be equal to the corresponding bit in the address value.

- One of the 16 possible PCI-X commands (`BX_CTSPLIT_CMDS`)
- The decoder (`BX_CTSPLIT_DEC`)

How to Program a Split Condition

To program a split condition:

1 Depending on the split decoder you want to program, set all split decoder properties for each decoder to default values with *BestXCTSplitCondDefaultSet*.

2 Define the request that should be given a split response by specifying the split decoder (0 … 3) and the split condition with *BestXCTSplitCondSet*.

3 Define the request queue where those requests will be put for later completion

Examples for Programming a Split Condition

Task Signal split response on an access to address range `0x100032XX` with memory read block.

Implementation

```
/* Set split decoder 0 … 3 to default values */
BX_TRY(BestXCTSplitCondDefaultSet(handle, 0));
BX_TRY(BestXCTSplitCondDefaultSet(handle, 1));
BX_TRY(BestXCTSplitCondDefaultSet(handle, 2));
BX_TRY(BestXCTSplitCondDefaultSet(handle, 3));

/* Set the address mask for split decoder 0 */
BX_TRY(BestXCTSplitCondSet(handle, 0, BX_CTSPLIT_ADDRMASK_LO,
                           0xFFFFFF00));
BX_TRY(BestXCTSplitCondSet(handle, 0, BX_CTSPLIT_ADDRMASK_HI,
                           0x00000000));
BX_TRY(BestXCTSplitCondSet(handle, 0, BX_CTSPLIT_ADDRVAL_LO,
                           0x10003200));
BX_TRY(BestXCTSplitCondSet(handle, 0, BX_CTSPLIT_CMDS,
                           BX_CTSPLIT_CMDS_READBLOCK));
BX_TRY(BestXCTSplitCondSet(handle, 0, BX_CTSPLIT_DEC,
                           BX_CTSPLIT_DEC_ANY));
```

Programming the Exerciser as a Completer-Initiator Device

A PCI-X completer-initiator device initiates PCI-X split completion transactions only. The completer-initiator can start a completion transaction only after the completer-target has given a split response.

To program the exerciser so that is acts as a PCI-X completer-initiator device, you have to:

- Set up generic completer-initiator properties. See *"Programming Generic Completer-Initiator Properties" on page 55*.
- Define how the completer-target intiates split completion transactions.

  See *"Programming the Completer-Initiator Behavior" on page 57*.

Programming Generic Completer-Initiator Properties

Generic completer-initiator properties are valid for a complete exerciser run.

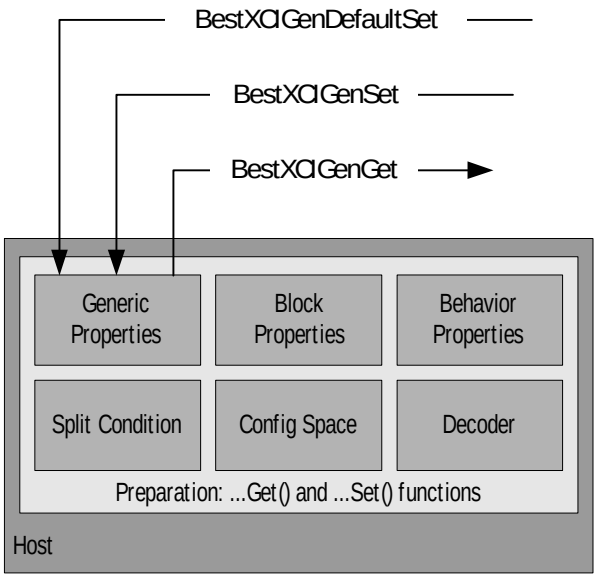
You can program the following generic requester-initiator properties:

- How many completer-initiator behaviors are used before the first one is used again (`BX_CIGEN_NUMBEH`)

  Valid values are 1 … 256.
- The content of the split completion message with `BX_CIGEN_MESSAGE_AD`. Set this property to a 32-bit value that defines the content of the split completion message as follows:
  - Bits 19 … 31 will be placed on the AD bus during the data phase.
  - Bits 0 … 11 are determined by the remaining byte count.
  - Bits 12 … 18 are reserved.

How to Program Generic Completer-Initiator Properties

The following figure shows the functions used to program the generic completer-initiator properties.



Programming Steps

To set generic completer-initiator property values on the host:

1. Set the whole generic completer-initiator properties memory to default values with *BestXCIGenDefaultSet*.
2. Set each property with *BestXCIGenSet* to the appropriate value.

   To get the value of one property, use *BestXCIGenGet*.

Programming the Completer-Initiator Behavior

The completer-initiator behaviors are stored in the completer-initiator behavior memory. Each memory entry holds one behavior; these are successively used for each request.

Each completer-initiator behavior has the following programmable properties:

- Select the request queue (`BX_CIBEH_QUEUE`) from which the next completion is generated:
  - Select the next queue (`BX_CIBEH_NEXT`)
  - Select no queue (`BX_CIBEH_NONE`)

    The requests will be accumulated for out-of-order completion.
  - Select any non-empty available queue (`BX_CIBEH_QAUTO`)
  - Select queue A, B, C or D with the option to skip the behavior if the currently selected queue is empty.
  - Select queue A, B, C or D with the option to wait until the currently selected queue gets a request, if this queue is empty.

NOTE Take care when using this option, because requests in other queues will not be completed if the selected queue does not receive a request.

- Select the size of the next partial completion transaction (`BX_CIBEH_PARTITION`):
  - The full byte count is transferred without disconnecting the completion (`BX_CIBEH_PARTITION_NO`).
  - The completion is disconnected at every N-th allowable disconnect boundary after the current start address. Valid values are 1 … 63.
- Split completion message (`BX_CIBEH_ERRMESSAGE`)

  If generation of a split completion message has been selected, a user-defined split completion message is generated, as opposed to a normal split completion transaction. The latter message, which would otherwise be generated, contains the data read or the default write completion message.

  For information on how to define the content of the message, see *"Programming Generic Completer-Initiator Properties" on page 55*.

NOTE Only one type of split completion message can be generated per test run.

- Wait for conditional start pattern (BX_CIBEH_CONDSTART)

  This property defines if the completion starts unconditionally or if a conditional start pattern must have occurred:

  - BX_CIBEH_CONDSTART_NO

    Unconditional start.

  - BX_CIBEH_CONDSTART_ONCE1

    After the exerciser has been started, block execution waits until the conditional start pattern 1 has occurred at least once.

  - BX_CIBEH_CONDSTART_WAIT1

    After the end of the previous requester-initiator sequence, block execution waits until the conditional start pattern 1 has occurred.

  - BX_CIBEH_CONDSTART_ONCE2

    After the exerciser has been started, block execution waits until the conditional start pattern 2 has occurred at least once.

  Pattern 1 and pattern 2 can be programmed with the Command Line Interface (CLI).

- Clock delay before assertion of REQ# (BX_CIBEH_DELAY)

  This property allows you to vary latencies between transactions. Sometimes the minimum achievable latency to the next completer-initiator transaction is restricted by the most recent event and sometimes by the data path configuration.

- Number of address steps (BX_CIBEH_STEPS)

  The number of address steps is the number of clock cycles between the assertion of GNT# and the assertion of FRAME# plus two clock cycles. These two clock cycles are designed into the register-to-register interface of PCI-X.

- 64-bit wide data transfer request (BX_CIBEH_REQ64)

- Release REQ# N clocks after the address phase (BX_CIBEH_RELREQ)

- Number of repeats (BX_CIBEH_REPEAT)

  The current behavior is repeated N times before the next behavior is used. Valid values are 0 … 256.

How to Program the Completer-Initiator Behavior

To program the completer-initiator behavior:

1 Set all entries of the completer-initiator behavior memory to default values.

Use *BestXCIBehDefaultSet*.

2 Program each behavior to one line of the completer-initiator behavior memory. Each behavior is specified by several behavior properties.

For each behavior property to be programmed, use *BestXCIBehSet*.

To query the value of a completer-initiator behavior property, use *BestXCIBehGet*.

3 Define how many behaviors should be executed.

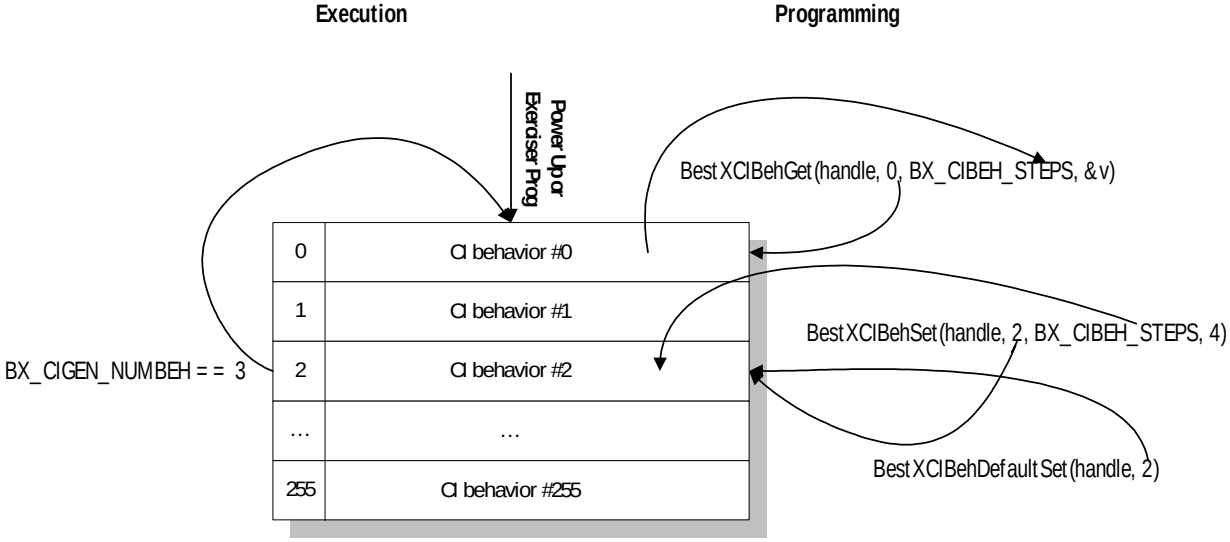
- First, set all generic completer-initiator properties to default values with *BestXCIGenDefaultSet*.
- Then, use *BestXCIGenSet* and set the completer-initiator generic property `BX_CIGEN_NUMBEH` to the appropriate value (`1 … 256`).

For more information about generic completer-target properties, see *"Programming Generic Completer-Initiator Properties" on page 55*.

4 Define how often the current behavior is applied before the next behavior is used.

Use *BestXCIBehSet* and set the completer-initiator generic property `BX_CIBEH_REPEAT` to the appropriate value. Valid values are 1 … 256.

The following figure shows the memory design, the available functions to program the completer-initiator behavior memory, and the execution order.



5 Download all exerciser settings and properties to the hardware with *BestXExerciserProg*.

Example for Programming the Completer-Initiator Behavior

Task Perform the following task:

- Set the completer-target behavior to the default.
- Program the E2929A to signal split response on each transfer.
- Delay the split completion for 100 clock cycles.
- Disconnect the completion at each ADB.

Implementation

```
/* Program completer-target behavior to default */

BX_TRY(BestXCTGenDefaultSet(handle));
BX_TRY(BestXCTBehDefaultSet(handle, 0));

/* Set up the split condition for split decoder 0 */

BX_TRY(BestXCTSplitCondDefaultSet(handle, 0));

BX_TRY(BestXCTSplitCondSet(handle, 0, BX_CTSPLIT_DEC,
                           BX_CTSPLIT_DEC_ANY));

BX_TRY(BestXCTSplitCondSet(handle, 0, BX_CTSPLIT_ADDRMASK_LO, 0));
BX_TRY(BestXCTSplitCondSet(handle, 0, BX_CTSPLIT_ADDRMASK_HI, 0));
BX_TRY(BestXCTSplitCondSet(handle, 0, BX_CTSPLIT_CMDS, 0xFFFFUL));

BX_TRY(BestXCTSplitCondSet(handle, 0, BX_CTSPLIT_QUEUE,
                           BX_CTSPLIT_QUEUE_NEXT));

/* Program the behavior properties to behavior memory line 0 */

BX_TRY(BestXCIGenDefaultSet(handle));

BX_TRY(BestXCIBehDefaultSet(handle, 0));

BX_TRY(BestXCIBehSet(handle, 0, BX_CIBEH_QUEUE,
                     BX_CIBEH_QUEUE_AUTO));

BX_TRY(BestXCIBehSet(handle, 0, BX_CIBEH_PARTITION, 1));
BX_TRY(BestXCIBehSet(handle, 0, BX_CIBEH_DELAY, 100));

BX_TRY(BestXExerciserProg(handle));
```

Programming the Exerciser as a Requester-Target Device

A PCI-X requester-target device is the target of a split completion transaction. It decodes PCI-X split completion transactions only.

To program the exerciser so that is acts as a PCI-X requester-target device, you have to:

- Set up the requester-target decoder.

  See *"Programming a Split Completion Decoder" on page 63*.

- Define how the requester-target reacts to transactions driven onto the bus.

  See *"Programming the Requester-Target Behavior" on page 63*.

- Define how many behaviors are used before the first one is used again.

  See *"Programming Generic Requester-Target Properties" on page 61*.

Programming Generic Requester-Target Properties

Generic requester-target properties are valid for a complete exerciser run.

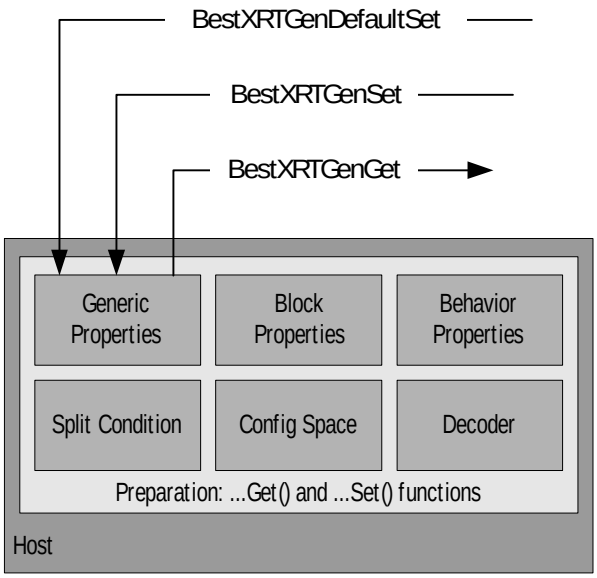
You can program the following generic requester-target property:

- How many requester-target behaviors are used before the first one is used again (`BX_RTGEN_NUMBEH`)

  Valid values are `1 … 256`.

How to Program Generic Requester-Target Properties

The following figure shows the functions used to program the generic requester-target properties.



Programming Steps

To set generic requester-target property values on the host:

1. Set the whole generic requester-target properties memory to default values with *BestXRTGenDefaultSet*.
2. Set each property with *BestXRTGenSet* to the appropriate value.

   To get the value of one property, use *BestXRTGenGet*.

Example for Programming Generic Requester-Target Properties

Task Program that three behaviors (0 … 2) are executed before the first one is used again.

Implementation

```
BX_TRY(BestXRTGenDefaultSet(handle));
BX_TRY(BestXRTGenSet(handle, BX_RTGEN_NUMBEH, 3));
```

Programming a Split Completion Decoder

To program the split completion decoder, you have to:

- Select the requester-target decoder.
- Specify the decoder size.
- Enable or disable the decoder.

This can be done with function *BestXTDecoderSet*.

Example

```
/* Enable the split completion decoder */
BX_TRY(BestXTDecoderSet(handle, BX_DEC_RT, BX_DECP_SIZE, 1));
```

Programming the Requester-Target Behavior

Each completer-initiator behavior has the following programmable properties.

- Decode speed A or B (`BX_RTBEH_DECSPEED`)

NOTE Decode speed A is only supported up to 66 MHz.

- Accept 64-bit wide data transfer (`BX_RTBEH_ACK64`)

  With this attribute, the target signals that it is capable of accepting 64-bit accesses:

  – 1: Assert the ACK64# signal

  – 0: Don't assert the ACK64# signal

- Initial target response (`BX_RTBEH_INITIAL`)

  This property defines how the completer-target responds after waiting the number of cycles specified under *Latency*.

- Number of initial latencies (`BX_RTBEH_LATENCY`)

  This property defines the number of wait cycles. Valid values are 3 … 34.

- Subsequent target response (BX_RTBEH_SUBSEQ)

  This property specifies the target response in subsequent data phases. It comes only into effect if the corresponding BX_RTBEH_INITIAL completer-target behavior was set to BX_RTBEH_INITIAL_ACCEPT.

- Subsequent data phases (BX_RTBEH_SUBSEQPHASE)

  If the value of behavior BX_RTBEH_SUBSEQ is set to BX_RTBEH_SUBSEQ_DISCONNECT, this property defines the subsequent data phase. Valid values are 0 … 2047.

- Number of repeats (BX_RTBEH_REPEAT)

  The current behavior is repeated N times before the next behavior is used. Valid values are 1 … 65536.

How to Program the Requester-Target Behavior

To program the requester-target behavior:

1 Set all entries of the requester-target behavior memory to default values.

  Use *BestXRTBehDefaultSet*.

2 Program each behavior to one line of the requester-target behavior memory. Each behavior is specified by several behavior properties.

  For each behavior property to be programmed, use *BestXRTBehSet*.

  To query the value of a requester-target behavior property, use *BestXRTBehGet*.

3 Define how many behaviors should be executed.

  – First, set all generic requester-target properties to default values with *BestXRTGenDefaultSet*.

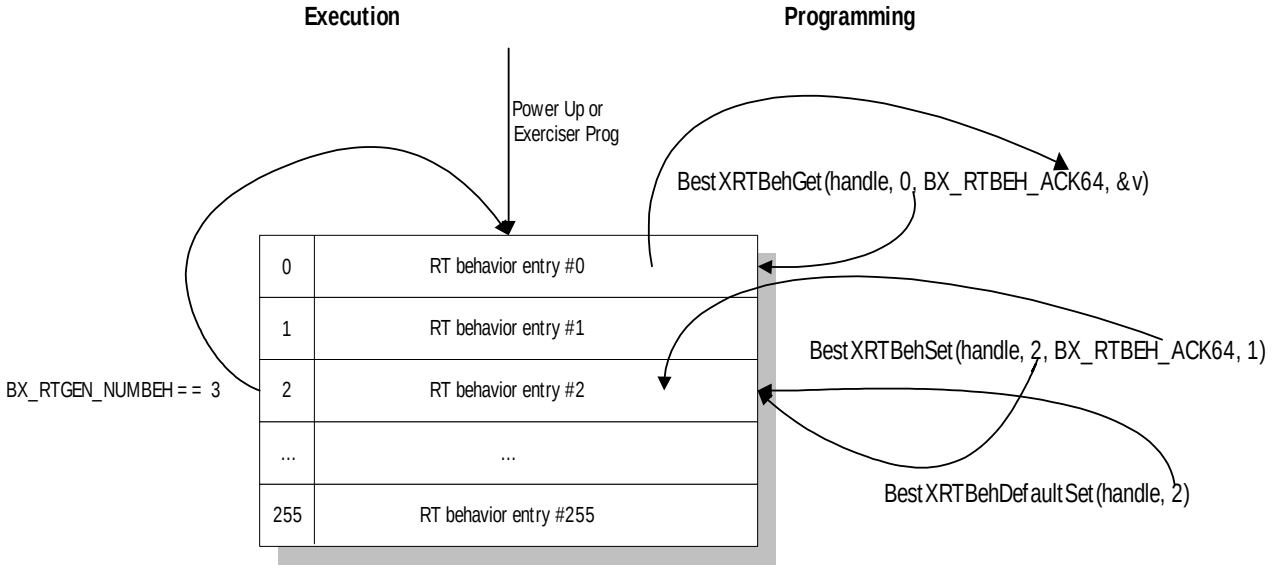
  – Then, use *BestXRTGenSet* and set the requester-target generic property BX_RTGEN_NUMBEH to the appropriate value (1 … 256).

  For more information about generic completer-target properties, see *"Programming Generic Requester-Target Properties" on page 61*.

4 Define how often the current behavior is applied before the next behavior is used.

  Use *BestXRTBehSet* and set the requester-target generic property BX_RTBEH_REPEAT to the appropriate value. Valid values are 1 … 65536.

The following figure shows the memory design, the available functions to program the requester-target behavior memory, and the execution order.



5 Download all exerciser settings and properties to the hardware with *BestXExerciserProg*.

Example for Programming the Requester-Target Behavior

Task Perform the following task:

- Set up the requester-target so that it uses decode speed A, B and C.
- Signal RETRY, then accept the transfer. Initial latency should be 12.

Implementation

```
/ * Program generic requester-target properties */
BX_TRY(BestXRTGenDefaultSet(handle));

BX_TRY(BestXRTGenSet(handle, BX_RTGEN_NUMBEH, 3));

/* Program the behavior properties decode speed and initial to
behavior memory line 0 */

BX_TRY(BestXRTBehDefaultSet(handle, 0));

BX_TRY(BestXRTBehSet(handle, 0,BX_RTBEH_DECSPEED,
                     BX_RTBEH_DECSPEED_A));

BX_TRY(BestXRTBehSet(handle, 0,BX_RTBEH_INITIAL,
                     BX_RTBEH_INITIAL_RETRY));
```

```
/* Program the behavior properties decode speed, initial and
latency to behavior memory line 1 */

BX_TRY(BestXRTBehDefaultSet(handle, 1));

BX_TRY(BestXRTBehSet(handle, 1,BX_RTBEH_DECSPEED,
                     BX_RTBEH_DECSPEED_B));

BX_TRY(BestXRTBehSet(handle, 1,BX_RTBEH_INITIAL,
                     BX_RTBEH_INITIAL_ACCEPT));

BX_TRY(BestXRTBehSet(handle, 1,BX_RTBEH_LATENCY, 12));

/* Program the behavior properties decode speed, initial and
latency to behavior memory line 2 */

BX_TRY(BestXRTBehDefaultSet(handle, 2));

BX_TRY(BestXRTBehSet(handle, 2,BX_RTBEH_DECSPEED,
                     BX_RTBEH_DECSPEED_B));

BX_TRY(BestXRTBehSet(handle, 2,BX_RTBEH_INITIAL,
                     BX_RTBEH_INITIAL_ACCEPT));

BX_TRY(BestXRTBehSet(handle, 2,BX_RTBEH_LATENCY, 12));

/* Download the settings to the testcard and run the exerciser */

BX_TRY(BestXExerciserProg(handle));
BX_TRY(BestXExerciserRun(handle));
```

Controlling the Exerciser

To control the generic behavior of the exerciser, you can program:

- The transaction scheduler

  The transaction scheduler controls how the exerciser schedules block transfers and split completions. See *"Scheduling Block Transfers and Split Completions" on page 68* for more information.

- The data generator

  The onboard data generator can be used to supply data patterns as an alternative to the data memory. See *"Programming the Data Generator" on page 73* for more information.

- The injection of errors

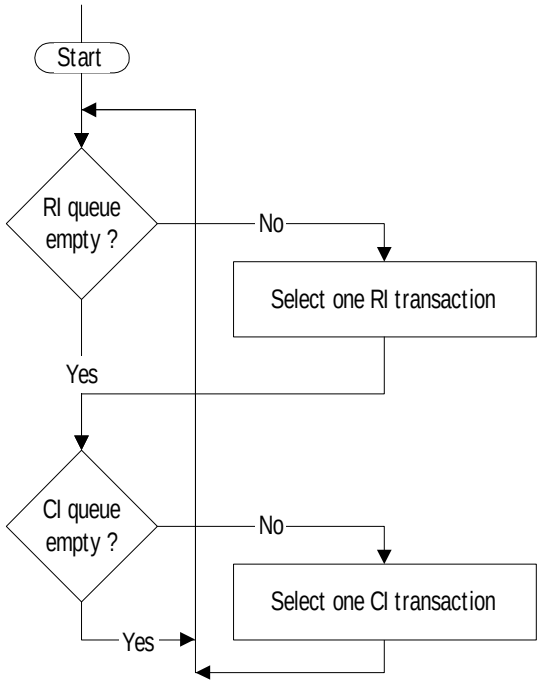
  Errors, such as wrong parity, PERR or SERR, can be injected in different phases of different resources. See *"Programming Errors Injection" on page 76* for more information.

Scheduling Block Transfers and Split Completions

For scheduling block transfers (requester-initiator transactions) and split completions (completer-inititator transactions), the software provides the following arbitration algorithms. The respective property values are given in parentheses.

- Automatical arbitration (`BX_EGEN_ARB_AUTO`)

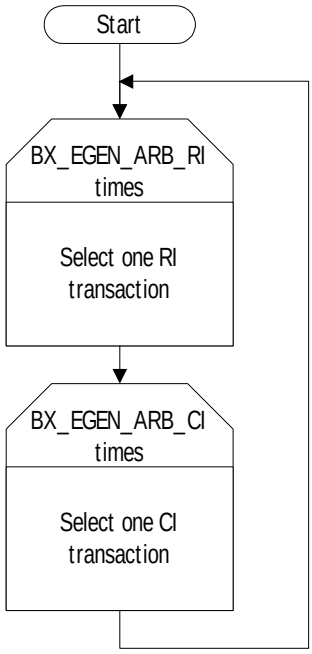
  The internal arbiter selects requester-initiator and split completion transactions one after the other. Empty queues are skipped.



- Constant arbitration (`BX_EGEN_ARB_CONST`)

NOTE Before you set this value, you can previously define a fixed number for requester-initiator and split completion transactions by setting values for `BX_EGEN_ARB_RI` and `BX_EGEN_ARB_CI`. Valid values are: `1(default) … 254`.

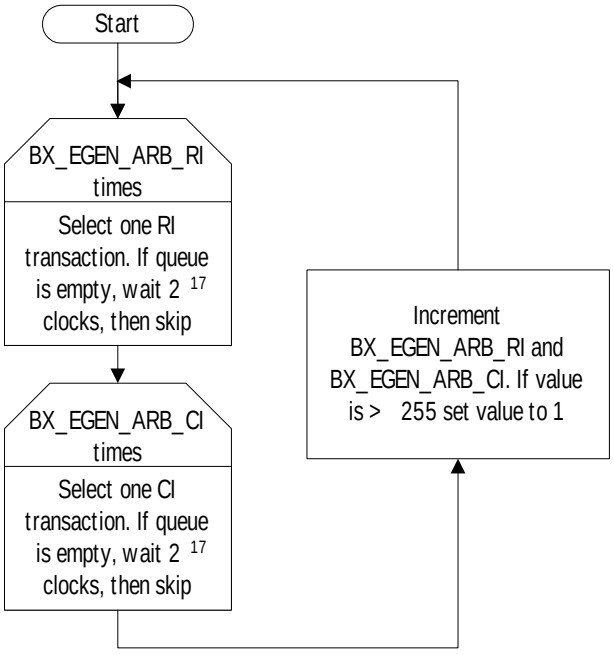
The arbiter first selects the defined number of requester-initiator transactions, then it selects the defined number of split completion transactions, then the fixed number of requester-initiator transactions again, and so on.



- Incremental arbitration (`BX_EGEN_ARB_INCREMENT`)

NOTE Before you set this value, you can previously define a fixed number for requester-initiator and split completion transactions by setting values for `BX_EGEN_ARB_RI` and `BX_EGEN_ARB_CI`. Valid values are: 1(default) … 254.

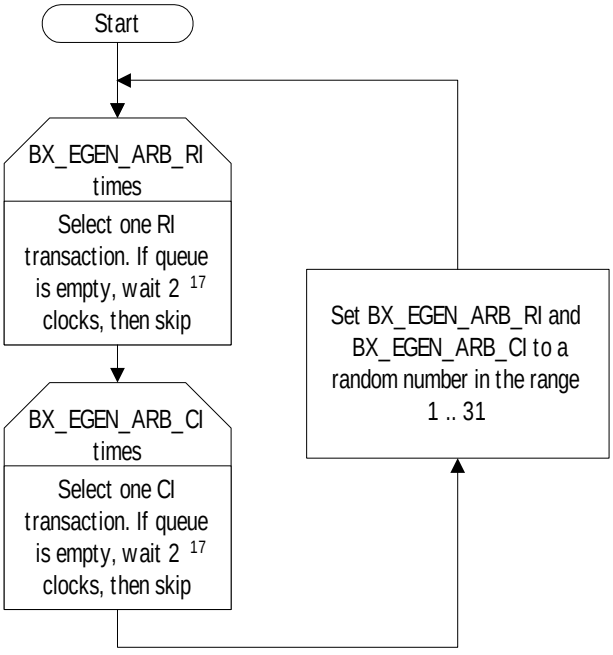
The arbiter first selects the defined number of requester-initiator transactions, then it selects the defined number of split completion transactions. If any queue is empty, the arbiter waits 2^{17} clocks before choosing the other queue. After one cycle, the values for `BX_EGEN_ARB_RI` and `BX_EGEN_ARB_CI` are incremented by one. After incrementing up to `255`, the value is set to `1`.



- Random arbitration (`BX_EGEN_ARB_RANDOM`)

  For each cycle, the software selects random numbers for `BX_EGEN_ARB_RI` and `BX_EGEN_ARB_CI` in the range `1 … 31`.

  For each cycle, the arbiter first selects the random number of requester-initiator transactions, then it selects the random number of split completion transactions. If any queue is empty, the arbiter waits 2^{17} clocks before choosing the other queue.



How to Schedule Block Transfers and Split Completions

To schedule block transfers and split completions, you have to set the arbiter property.

Programming Steps

To set the arbiter property values on the host:

1. Set all exerciser generic properties to default values with *BestXExerciserGenDefaultSet*.
2. Regarding the algorithm you want to program, you can first define fixed values for the number of requester-initiator and completer-initiator transactions by setting `BX_EGEN_ARB_RI` and `BX_EGEN_ARB_CI` with *BestXExerciserGenSet*.

   This only has an effect when programming the constant or the incremental algorithm.
3. Program the algorithm by setting the `BX_EGEN_ARB` property with *BestXExerciserGenSet* to the appropriate value.

   For appropriate values, see *"Scheduling Block Transfers and Split Completions" on page 68*.

   To get the value of one property, use *BestXExerciserGenGet*.
4. Write the property to the testcard with the *BestXExerciserProg*.

Example for Scheduling Block Transfers and Split Completions

Task

Program the internal arbiter, so that the number of requester-initiator and split completion transactions executed sequentially are incremented with each cycle. Set the number of requester-initiator and completer-initiator transactions to 5.

Implementation

```
/* Program the Exerciser generic properties */

BX_TRY(BestXExerciserGenSet(handle, BX_EGEN_ARB_RI, 5));
BX_TRY(BestXExerciserGenSet(handle, BX_EGEN_ARB_CI, 5));
BX_TRY(BestXExerciserGenSet(handle, BX_EGEN_ARB, BX_EGEN_ARB_INCR));

BX_TRY(BestXExerciserProg(handle));
```

Programming the Data Generator

Data Generator Features

You can use the onboard data generator to supply data as an alternative to using the data memory. The data generator has the following features:

- It allows the testcard to deliver fast data patterns without initial latencies.
- It generates unique data patterns (up to 2^{21} DWORDs) that allow you to deterministically link a certain address to a certain data pattern. This feature allows you a unidirectional data path verification. For more information about the unidirectional data path verification, please refer to *Agilent E2929A/B Opt. 300 PCI-X Exerciser User's Guide*.

Furthermore, the data generator provides:

- 21-bit width of unique data. The data itself is 64 bits wide.
- A programmable start value that can be used to generate unique data patterns.
- That the data pattern can be changed with every data phase.

Data Generator Properties

You can program the following data patterns. The respective properties are given in parentheses.

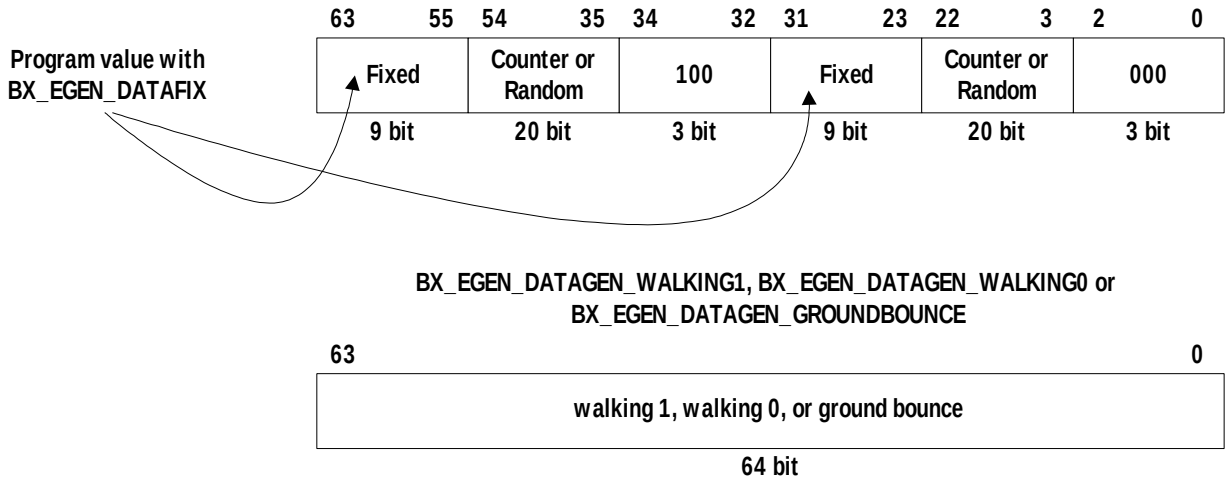
- Count-up data pattern (unique data) (`BX_EGEN_DATAGEN_COUNTER`)

  To generate a count-up data pattern, you also have to specify an offset to the bus address (`BX_EGEN_DATASEED`). Valid values are `0 … (2^20 - 1)`.

  The counter creates a 64-bit wide data pattern, consisting of two count values from bit `0 … 22` and from bit `32 … 54`. The remaining 18 bits can be preset with an arbitrary value or the identification to enable an easy identifier for any seen data in the system.

  If the counter is programmed with an initiator identification (`BX_EGEN_DATAFIX_MASTERID`), the data compare (`BX_EGEN_PARTCOMP`) can be switched off for the 18 fixed bits to still allow unidirectional data verification.

The bit assignment of the data generator is as follows:



- Generating walking ones or zeros (`BX_EGEN_DATAGEN_WALKING1`, `BX_EGEN_DATAGEN_WALKING0`)
- Generating a pseudo-random pattern (unique data) (`BX_EGEN_DATAGEN_COUNTMIX`)

  To generate a pseudo-random pattern, you also have to specify an offset to the bus address (`BX_EGEN_DATASEED`). Valid values are `0 … (2^20 - 1)`.

  The generated data is a 64-bit wide pattern that changes with a period of 2^{21} and appears as a pseudo-random sequence.

- Generating a ground bounce pattern (`BX_EGEN_DATAGEN_GROUNDBOUNCE`)

  Generate data where `0x00000000` and `0xffffffff` patterns alternate.

NOTE This pattern can be used for unidirectional data path verification. For more information, please refer to *Agilent E2929A/B Opt. 300 PCI-X Exerciser User's Guide*.

How to Program the Data Generator

Programming Steps

To program the data generator, you have to perform the following steps:

1. Ensure that the data generator was defined as data source for requester-initiator block transfers or the target decoder. To program this, see *"How to Program Block Transfers" on page 31* and *"How to Program a Decoder" on page 43*.
2. Set the whole exerciser generic properties to default values with *BestXExerciserGenDefaultSet*.
3. Regarding the pattern you want to program, you can define a fixed value for the data generator or an initiator ID. Set `BX_EGEN_DATAFIX` to a 18-bit value or to `BX_EGEN_DATAFIX_MASTERID` with *BestXExerciserGenSet*.

   This only has an effect when programming the counter pattern and the pseudo random pattern.
4. Program the pattern by setting the `BX_EGEN_DATAGEN` property with *BestXExerciserGenSet* to the appropriate value.

   For valid values, see *"Programming the Data Generator" on page 73*.

   To get the value of one property, use *BestXExerciserGenGet*.
5. Write the settings to the testcard with the *BestXExerciserProg*.

Example for Programming the Data Generator

Task

Program the data generator to use the walking 0 datapattern.

Implementation

```
BestXExerciserGenSet(handle, BX_EGEN_DATAGEN,
                     BX_EGEN_DATAGEN_WALKING0);
```

Programming Errors Injection

Programming errors takes place in the following steps.

1. Specifying the resource from where the error is injected.
2. Specifying the exact location within that resource.
3. Specifying the type of error.
4. Specifying the phase in which the exceptions are generated.

Restrictions on Errors Injection

Errors injection is subject to the following restrictions:

- The injection of wrong parity (32/64) will not occur under the following conditions:
  - For any device acting as a receiver of data. This is normal behavior because it is the device driving the bus that is responsible for generating and drive parity, not the receiver. That is why address phase parity generation works but not data phase parity generation.
  - On all write commands in first data phase with target decode speed A and zero initial waits.
- The injection of wrong PERR will not occur:
  - For any device acting as a sender of data. This is normal behavior because it is the device that receives the data that is responsible for asserting PERR.
  - During/due to an address phase or attribute phase. See PCI-X specification section 5.4.3.
- Attribute phase error injection is not supported.

Table of Supported Actions The following table lists all actions supported by the software:

NOTE **Yes** means that this error can be injected.

No means that this error cannot be injected.

BX_EGEN_ ERR_SOURCE	BX_EGEN_ ERR_PHASE	BX_EGEN_ERR_ WRPAR	BX_EGEN_ERR_ WRPAR64	BX_EGEN_ERR_PERR	BX_EGEN_ERR_SERR
RI Block	Addr	Yes	Yes	No	Yes
	Attr	No	No	No	No
	Data read	No	No	Yes (PERR is only generated by data receivers.)	Yes
	Data write	Yes (PAR is only generated by data senders.)	Yes (PAR64 is only generated by data senders.)	No	No
RI Behavior	Addr	Yes	Yes	No	Yes (Does not appear to observe specified behavior line. Occurs on every transfer.)
	Attr	No	No	No	No
	Data read	No	No	Yes	Yes
	Data write	Yes	Yes	No	Yes
CT Behavior	Addr	No (The RI is responsible for generating parity.)	No (The RI is responsible for generating parity.)	No	Yes
	Attr	No	No	No	No
	Data read	Yes (Only on data phase 1.)	Yes (Only on data phase 1.)	Yes (Only on data phase 1.)	Yes (Only on data phase 1.)
	Data write				
CI Behavior	Addr	Yes (With mem read from RI.)	Yes (With mem read from RI.)	No	Yes (3 clocks after address phase.)
	Attr	No	No	No	No
	Data read	Yes	Yes	No (The CI is always the data sender, and senders do not generate PERR.)	Yes (3 clocks after data phase.)
	Data write				

BX_EGEN_ ERR_SOURCE	BX_EGEN_ ERR_PHASE	BX_EGEN_ERR_ WRPAR	BX_EGEN_ERR_ WRPAR64	BX_EGEN_ERR_PERR	BX_EGEN_ERR_SERR
RT Behavior	Addr	No (The CI is responsible for generating parity.)	No (The CI is responsible for generating parity.)	No	Yes (Also puts SERR when RI is active.)
	Attr	No (The CI is responsible for generating parity.)	No (The CI is responsible for generating parity.)	No	No
	Data read	No (The RT is always the data receiver, and receivers do not generate PAR/PAR64.)	No (The RT is always the data receiver, and receivers do not generate PAR/PAR64.)	Yes (3 clocks after the specified data phase.)	Yes (2 clocks after the specified data phase.)
	Data write				
Decoder	Addr	No (The decoder has not yet claimed the transaction.)	No (The decoder has not yet claimed the transaction.)	No (The decoder has not yet claimed the transaction.)	No
	Attr				
	Data read	Yes	Yes	Yes	Yes
	Data write				

How to Program Errors Injection

Programming Steps

To program the data generator, you have to perform the following steps:

1. Set the whole exerciser generic properties to default values with *BestXExerciserGenDefaultSet*.
2. Specify whether and from which resource an error is injected by setting `BX_EGEN_ERR_SOURCE` with *BestXExerciserGenSet* to the appropriate value.

   Valid values are:

   - Requester-initiator block memory (`BX_EGEN_SOURCE_RIBLK`)
   - Requester-initiator behavior memory (`BX_EGEN_SOURCE_RIBEH`)
   - Completer-target behavior memory (`BX_EGEN_SOURCE_CTBEH`)
   - Decoder address range (`BX_EGEN_SOURCE_DEC`)
   - Completer-initiator behavior memory (`BX_EGEN_SOURCE_CIBEH`)
   - Requester-target behavior memory (`BX_EGEN_SOURCE_RTBEH`)

3 Specify the exact location within the selected resource by setting `BX_EGEN_ERR_NUM` with *BestXExerciserGenSet* to the appropriate value. The valid value depends on the selected resource.

The exact location depends on the selected resource. Please refer to *Agilent E2929A/B Opt.320 C-API/PPR Programming Reference* or *Agilent E2922A/B Opt.320 C-API/PPR Programming Reference* for more information.

To get the current value of the property, use *BestXExerciserGenGet*.

4 Specify the type of error to be injected.

Use *BestXExerciserGenSet* and set the following properties:

– `BX_EGEN_ERR_WRPAR` (Wrong parity (PAR))

0: Parity is generated correctly; 1: Parity is inverted.

– `BX_EGEN_ERR_WRPAR64` (Wrong parity, 64-bit (PAR64))

0: Parity is generated correctly; 1: Parity is inverted.

– `BX_EGEN_ERR_PERR` (Parity error (PERR))

0: PERR is not asserted; 1: PERR# is asserted..

– `BX_EGEN_ERR_SERR` (System error (SERR))

0: SERR is not asserted; 1: SERR# is asserted.

To get the current value of the property, use *BestXExerciserGenGet*.

5 Specify in which phase (address or attribute) the possible exceptions are generated.

Use and set `BX_EGEN_ERR_PHASE` to:

– `BX_EGEN_ERR_PHASE_ADDR` (address phase)

– `BX_EGEN_ERR_PHASE_ATTR` (attribute phase)

– `1 … 512 (1024 in 32 bit systems)` (data phase)

To get the current value of the property, use *BestXExerciserGenGet*.

6 Write the settings to the testcard with *BestXExerciserProg*.

Example for Programming Errors Injection

Task Set up the E2929A so that it generates wrong parity on the 32-bit lines during the attribute phase of block transfer #2.

Implementation

```
BX_TRY(BestXExerciserGenSet(handle, BX_EGEN_ERR_SOURCE,
                            BX_EGEN_ERR_SOURCE_RIBLK));
X_TRY(BestXExerciserGenSet(handle, BX_EGEN_ERR_NUM, 1));
BX_TRY(BestXExerciserGenSet(handle, BX_EGEN_ERR_PHASE,
                            BX_EGEN_ERR_PHASE_ATTR));
BX_TRY(BestXExerciserGenSet(handle, BX_EGEN_ERR_WRPAR, 1));
BX_TRY(BestXExerciserProg(handle));
```

Programming the Expansion ROM

The expansion ROM is typically used as boot ROM and can contain a power-on-self-test, BIOS and interrupt service routines.

The expansion ROM of the testcard features “code-in-place execution” (XIP) (instead of copying the expansion ROM content into system memory for execution). This is beyond PCI-X specification, but can be used in a system in which system memory does not yet work.

The expansion ROM of the testcard is accessible:

- By means of C-API functions to fill and read the expansion ROM contents:
  - To fill the expansion ROM content, use *BestXExpRomWrite*.
  - To read the expansion ROM content, use *BestXExpRomRead*.
- Through a memory range defined in the “expansion ROM base address register” in the testcard’s configuration space.

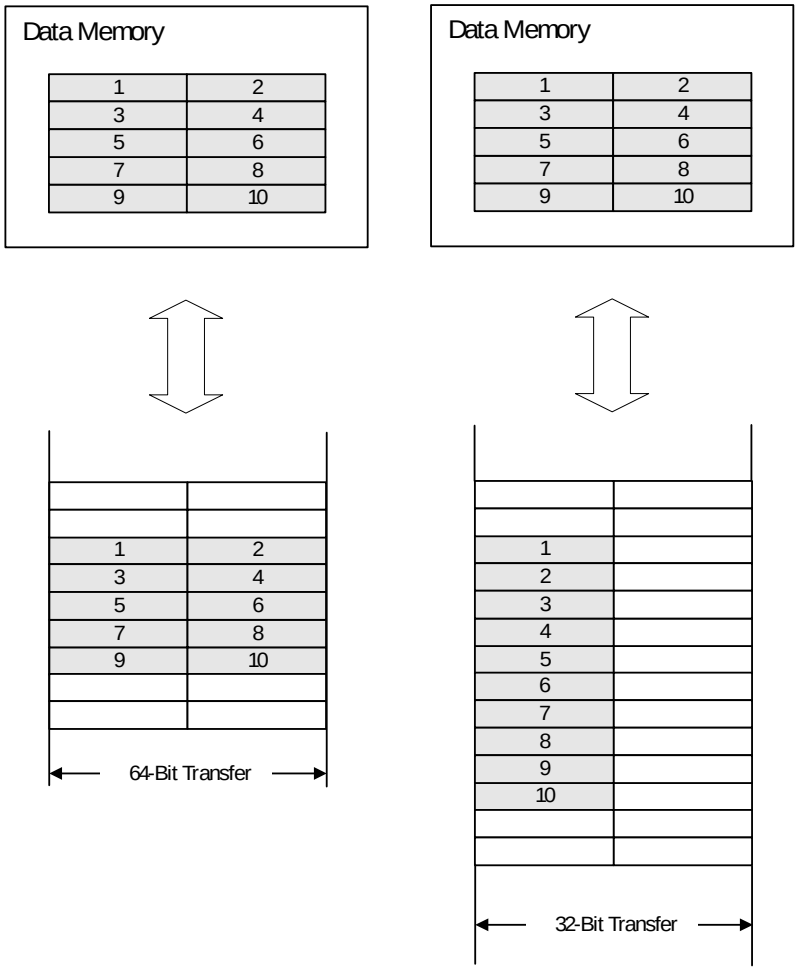
Programming the Data Memory

To read and to fill the data memory of the testcard with compare data and data to send, you need to access the testcard's data memory from your control PC.

Data Alignment

The testcard provides a 1-MB (128K × 2 dwords) programmable read/write data memory. For usability reasons, when transferring data between the PCI-X bus under test and the data memory, the data must be aligned with respect to the PCI-X bus width and the width of data transfer (32-bit or 64-bit).

The following figure illustrates how data that is stored in the data memory is driven onto the PCI-X bus and how received data is stored (for 64-bit and 32-bit data transfer).



To align the data correctly for data transfers to or from the PCI-X bus, follow these rules:

- For 64-bit transfers:

  The specified byte addresses must be 64-bit aligned. For both the internal address and the PCI-X bus addresses, the least significant **three bits** must be 0.

- For 32-bit transfers:

  The specified byte addresses must be 32-bit aligned. For both the internal address and the PCI-X bus addresses, the least significant **two bits** must be 0. The third bit of the internal and external address must be equal.

Initiator and Target Partitions

Basically, you are free to use any part of the data memory for the initiator and the target. However, if you need to use the data memory as a resource for all devices at the same time, it is recommended to define memory partitions.

This can be done by using different internal addresses for the initiator and the target. This applies to the following parameters:

- For the initiator:

  The block properties `BX_RIBLK_INTADDR` and `BX_RIBLK_NUMBYTES` must be set appropriately.

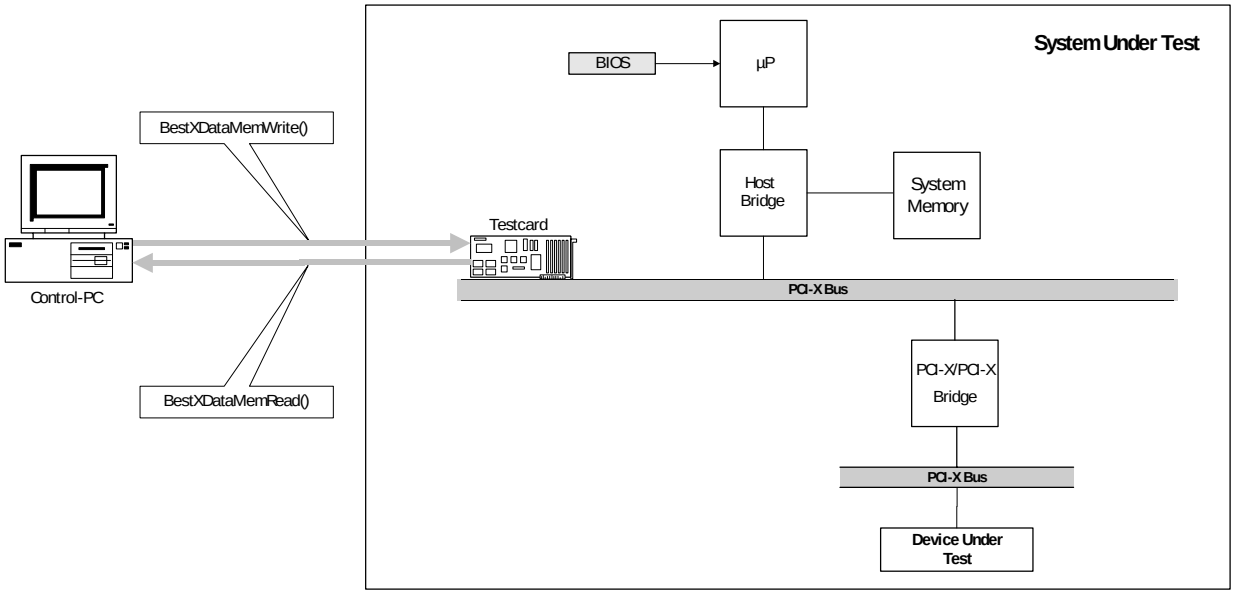
- For the target:

  When you set the `BX_DECP_RESOURCE` property to `BX_DECP_RESOURCE_MEM` (data memory) or data compare is switched on (you set the `BX_DECP_COMPARE` property to 1), the properties `BX_DECP_RESBASE` address and `BX_DECP_RESSIZE` must be set appropriately.

NOTE If the initiator of the testcard is communicating with its own target via the PCI-X bus, the target has no access to the data memory. When the initiator performs write commands, the target cannot store data in the data memory.

How to Program the Data Memory

The following figure shows functions used to program the data memory.



Programming Steps

Programming the data memory requires the following steps:

- Perform data transfer to and from the internal data memory.
  - To write data to the internal data memory of the testcard via the control interface, the control PC runs the C program and can be used to generate the data to be written.

    Use *BestXDataMemWrite*.
  - To read data from the testcard, the same method is used in reverse.

    Use *BestXDataMemRead*.

Example for Programming the Data Memory

Task

Read a memory block of 32 Kbyte (0x8000) from the data memory of the testcard, beginning with internal address 0x0000, to the control PC memory (specified by `buffer`).

Implementation

```
bx_int8 buffer[32*1024];

...

BX_TRY(BestXDataMemInit(handle));
BX_TRY(BestXDataMemRead(handle, 0x0000, 0x8000, buffer));
```

Programming Data Transfer To and From the Host

The testcard provides *host access functions* to access the registers of a PCI-X device. These functions allow you to access the memory, I/O and configuration space of a device.

Reading Data

To read the value from a specific PCI-X device register in a 32-bit address space to the control PC, use *BestXHostPCIRegRead*.

The bus address is a byte address. This function performs single-cycle transactions and sets automatically the correct byte enables corresponding to the word size and bus address.

Writing Data

To write the value from the control PC to a specific PCI-X device register in a 32-bit address space, use *BestXHostPCIRegWrite*.

The bus address is a byte address. This function performs single-cycle transactions and sets automatically the correct byte enables corresponding to the word size and bus address.

Example for Host Access

Task

Transfer a dword from a register in a PCI-X device at the physical memory address 0x8000 to the control PC.

Implementation

```
BX_TRY(BestXHostPCIRegRead(handle, BX_ADDRSPACE_MEM, 0x8000,
BX_SIZE_DWORD));
```

Programming PCI-X Interrupts

The testcard can generate any PCI-X interrupt INTA# ... INTD#. Programming interrupts queries the following steps:

1. The interrupts must be generated.
2. The interrupt status has to be queried.
3. After the interrupt status has been queried, the interrupts must be cleared.

It is necessary to be able to generate interrupts, for example, when developing interrupt drivers for a PCI-X device.

How to Generate PCI-X Interrupts

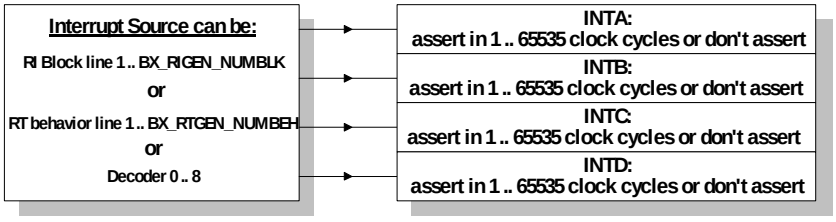
To program an interrupt:

1 Generate an PCI-X interrupt INTA# ... INTD# with *BestXInterruptGenerate*.

– or –

Generate an interrupt via the hardware if a certain requester-initiator block or requester-target behavior line is executed or if a certain decoder is accessed.

The following figure shows the sources and when the interrupt is asserted.



– Specify the **source** for the interrupt.

Use *BestXExerciserGenSet* and set the generic exerciser property `BX_EGEN_INT_SOURCE` to the required value (none, RIBLK, RTBEH, or decoder).

– Specify the **exact location** within the source.

Use *BestXExerciserGenSet* and set the generic exerciser property `BX_EGEN_INT_NUM` to the required value (Position in RIBLK or RTBEH memory or decoder number (0 .. 8)).

– Specify the **delay** in clock cycles for INTA … INTD after the event that triggers the interrupt has occurred.

Use *BestXExerciserGenSet* and set the generic exerciser properties `BX_EGEN_INT_DELAYA` … `BX_EGEN_INT_DELAYD` to the required values (`BX_EGEN_INT_DELAY_NO` or `1 … 65535`).

2 Write the exerciser generic properties to the testcard with *BestXExerciserProg*.

3 Query the interrupt status.

Use *BestXStatusRead*.

Refer to *bx_statustype* in the *Agilent E2929A/B or E2922A/B Opt. 320 C-API/PPR Programming Reference* to see the properties and values to be get.

4 Clear the interrupt.

Use *BestXStatusClear*.

Refer to *bx_statustype* in the *Agilent E2929A/B or E2922A/B Opt. 320 C-API/PPR Programming Reference* to see the properties to be cleared.

Interrupt Status Register

The interrupt status register is located in the private section of the testcard's configuration space. It can be read, for example, by interrupt drivers to determine whether the testcard has generated an interrupt.

The offset within the configuration space is **0x51**. The bits of interrupt status register are explained in the table below.

Bit	Oper.	Value	Meaning
0	Interrupt A pending.		
	Read	0	No interrupt pending.
		1	An interrupt has been generated by the testcard and is waiting to be serviced.
	Write	1	The appropriate interrupt is cleared.
1	Interrupt B pending. (Read/write operation, value, meaning see interrupt A.)		
2	Interrupt C pending. (Read/write operation, value, meaning see interrupt A.)		
3	Interrupt D pending. (Read/write operation, value, meaning see interrupt A.)		

Example for Programming PCI-X Interrupts

Task Program the following:

- Let the E2929A assert INTB 200 clock cycles after block transfer #1 was started.
- Query the interrupt after BestXExerciserRun.
- After that, clear the interrupt.

Implementation

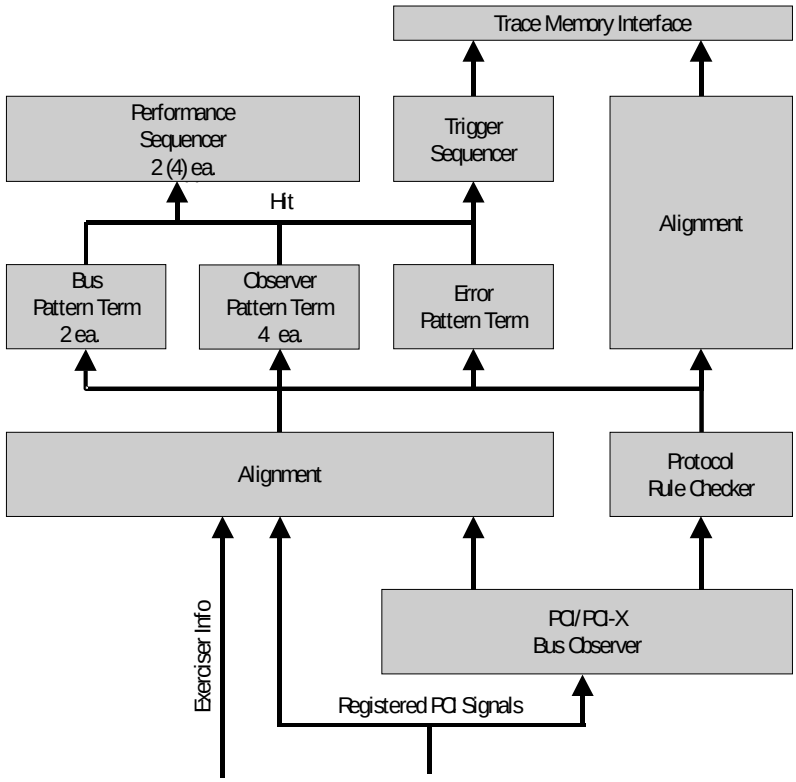
```
#include <xpciapi.h>

int main(int argc, char* argv[])
{
   BX_TRY_VARS_NO_PROG;

   /* additional local variable declarations here */
   bx_handletype handle;

   BX_TRY_BEGIN

   {

      BX_TRY(BestXOpen(&handle, BX_PORT_RS232, BX_PORT_COM1));
      BX_TRY(BestXRS232BaudRateSet(handle, BX_BD_57600));
      BX_TRY(SetupForInterrupt(handle));

      /* Specify the source for the interrupt */

      BX_TRY(BestXExerciserGenSet(handle, BX_EGEN_INT_SOURCE,
      BX_EGEN_INT_SOURCE_RIBLK));

      /* Specify the location within the source */

      BX_TRY(BestXExerciserGenSet(handle, BX_EGEN_INT_NUM, 1));

      /* Specify the delay the interrupt is asserted within the
      source */

      BX_TRY(BestXExerciserGenSet(handle,BX_EGEN_INT_DELAYB,200));

      /* Write the settings to the testcard and run the exerciser*/

      BX_TRY(BestXExerciserProg(handle));
      BX_TRY(BestXExerciserRun(handle));

      /* Query the interrupt */

      bx_int32 val;
      BX_TRY(BestXStatusRead(handle, BX_STAT_INTB, &val));
      printf("the value of the number.. before %d\n",val);
```

```
      /* Clear the interrupt */

      BX_TRY(BestXStatusClear(handle, BX_STAT_INTB));
      BX_TRY(BestXStatusRead(handle, BX_STAT_INTB, &val));

      printf("the value of the number.. after  %d\n",val);
      BX_TRY(BestXClose(handle));

   }

   BX_TRY_CATCH
   {
      // cleanup, if necessary
      printf("%s\n", BestXErrorStringGet(BX_TRY_RET));
   }

   BX_ERRETURN(BX_TRY_RET);
}
```

Programming the Analyzer

The tasks of PCI-X analysis are to monitor the PCI-X bus, to detect specific events, and to measure and evaluate the occurrences of signals on the bus.

The following figure shows the components of the analyzer.



The following sections explain how to program the testcard's analyzer to fulfill the different tasks:

- *"Programming the Protocol Observer" on page 90* explains how to mask individual protocol rules and how to read the observer result registers.
- *"Programming Pattern Terms" on page 94* explains all types of pattern terms, and how to use and program them.

- *"Programming the Trigger Sequencer" on page 96* explains how to program the sequencers.

  Basically, all sequencers on the testcard work in the same manner. There are many parameters controlling the sequencers. The principles of the sequencers are explained, and an example of using the trace memory trigger sequencer is provided.

- *"Programming the Trace Memory" on page 104* explains how to use the trace memory and how to program its sequencer and the storage qualifier. Information about how to upload and evaluate the contents of the trace memory is also provided.

- *"Programming the Performance Sequencer" on page 111* explains how to program the performance measures.

Programming the Protocol Observer

The protocol observer monitors 53 different protocol rules simultaneously. The protocol rules refer to PCI-X specification rules. An "any error" output for triggering purposes is provided, as well as registers to latch the first occurring errors and the accumulating subsequent errors.

Error Register Contents

The protocol observer provides two error registers containing:

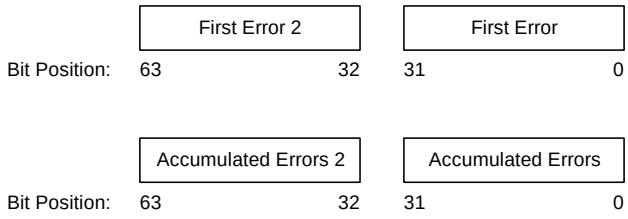
- Bits for the protocol rule violations that have occurred first. Often the first rule violations are the reason for subsequent rule violations.

  Each individual rule can be masked from being detected as "first rule violation". This allows you to exclude certain rule violations from triggering the analyzer.

- A flag bit for each rule violated during observation.

Error Register Design

Both of the following registers hold a flag bit for each rule and, therefore, consist of two registers each with a length of 32 bits.



The contents of the error registers can be read by means of the testcard C-API, which converts them into a text string that describes the violated rules.

Further Use

A detected protocol violation can:

- Be used as input for pattern terms (see *"Programming Pattern Terms" on page 94*).
- Trigger the trace memory (see *"Programming the Trace Memory" on page 104*).

  The rule violation(s) cause a "bus error", which can be used as a trigger signal. It is aligned to the first clock at which the error was detected.

TIP

This holds true except for parity errors: they are aligned to the transfer cycle where the data does not match the PAR signal. A storage qualifier can be used to store only the incorrect data phases.

How to Program the Protocol Observer

The testcard's programming interface provides functions for programming the protocol observer.

Programming Steps

Programming the protocol observer requires the following steps:

1. Check if the protocol observer has detected a protocol error by reading the card status (`BX_STAT_OBS_ERR`) with *BestXStatusRead*.
2. If an protocol error has occurred (observer status = 1), reading a text string containing all errors that were found in the first error register and the accumulated error registers.

   Use *BestXObsStatusRead*.

With the Agilent E2929A/B testcard, you can trigger on particular PCI-X protocol errors and view the captured waveforms. To perform this:

1 Optionally, mask out all rules that you are not interested in.

Use *BestXObsMaskProg*.

2 Program an observer pattern term.

See *"How to Program Pattern Terms" on page 94*.

3 Program the trigger sequencer.

See *"How to Program the Trigger Sequencer" on page 99*.

4 View the captured waveforms.

See *"How to Program the Trace Memory" on page 105*.

Example for Programming the Protocol Observer

Task Read the occurred protocol rule violations.

Implementation

```
#include "stdafx.h"
#include <xpciapi.h>
#include "SetupUtil.h"

int main(int argc, char* argv[])
{
   BX_TRY_VARS_NO_PROG;

   /* additional local variable declarations, here */
   bx_handletype handle;
   BX_TRY_BEGIN
   {
      /* Open the connection to the card */
      BX_TRY(BestXOpen(&handle, BX_PORT_RS232, BX_PORT_COM2));
      BX_TRY(BestXRS232BaudRateSet(handle, BX_BD_57600));
      printf("Opened the connection on the card\n");

      bx_int32 erroroccured;

      /* Query if any protocol errors occurred */

      BX_TRY(BestXStatusRead(handle, BX_STAT_OBS_ERR,
                             &erroroccured));

      if (erroroccured == 0)
      {
         printf("No protocol errors were detected\n");
      }
      else
      {
         printf("PROTOCOL ERROR HAS OCCURRED\n");

         /* Print the text string of all errors that occurred */
         bx_charptrtype errortext;
         BX_TRY(BestXObsStatusRead(handle, &errortext));

         printf("The status is %s\n", errortext);
      }
      BX_TRY(BestXClose(handle));
      printf("Closed the connection to the card\n");
   }
   BX_TRY_CATCH
   {
   /* cleanup, if necessary */
   printf("%s\n", BestXErrorStringGet(BX_TRY_RET));
   }
   return 0;
}
```

Programming Pattern Terms

The pattern terms are programmed using logical equations that define the pattern to be recognized. Each pattern term is identified by its pattern term identifier. For a list of valid pattern term identifiers, see "*bx_patttype*" in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference*.

The pattern terms are programmed by means of signals and logical operators.

Using Pattern Terms

The pattern terms (also known as pattern recognizers) compare bus states with programmable conditions. Their output (1 = pattern found, 0 = pattern not found) can be used:

- As input for sequencers, for example, the trace memory trigger sequencer (see *"Programming the Trigger Sequencer" on page 96*).
- For storage qualification for the trace memory (see *"Programming the Trace Memory" on page 104*).
- When counting bus events for performance analysis (see *"Programming the Performance Sequencer" on page 111*).
- For requester-initiator conditional start based on the detection of a specific event on the PCI-X bus.

As input, the pattern terms can use all the signals specified in "*bx_signaltype*" in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference*.

On the testcard, the following pattern terms are implemented:

- Two bus pattern terms (bus0, bus1)
- Four observer patterns terms (obs0 … obs3)
- One error pattern term (err0)
- Two conditional pattern terms (cond0, cond1)

How to Program Pattern Terms

Programming Steps

To specify a pattern term, use *BestXPattProg*.

This pattern term can be used in the condition strings of a sequencer description table.

Example for Programming Pattern Terms

Task Program the following two patterns:

- Program bus pattern `(bus0) to` detect an address phase that addresses the range 10003000\h … 10003fff\h.
- Program an observer pattern to detect an active bus state.

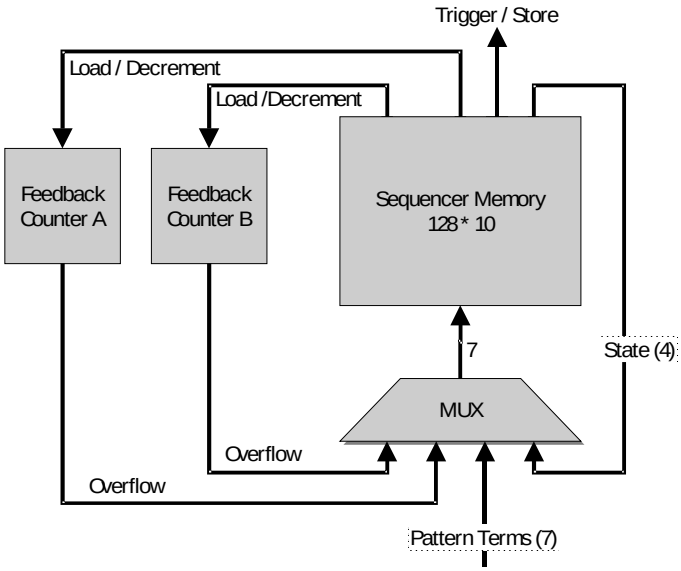
Implementation

```
/* Program the bus pattern */

BX_TRY(BestXPattProg(handle, BX_PATT_BUS0, "AD32==10003xxx\\h &&
                                              addr_phase==1"));

/* Program the observer pattern. For bstate, the numeric value for
an active bus state is "1". See "bx_signaltype" in the
Agilent E2929A/B Opt. 320 C-API/PPR Reference. */

BX_TRY(BestXPattProg(handle, BX_PATT_OBS0, "bstate == 1"));
```

Programming the Trigger Sequencer

The sequencers of the testcard detect bus state sequences. The sequencers use programmable pattern terms to compare bus states with programmable conditions.

The following figure shows the hardware components of the trigger sequencer.



All sequencers provide an internal memory, state machine, and two 32-bit feedback counters (A and B). The statemachine controls the operation of the sequencer. The sequencer has 7 input registers. These registers can be used for input from pattern terms and for state feedback from the sequencer output. A maximum of 2^4= 16 states is the practical limit—because at least one pattern term is always needed.

Setting up the Sequencer

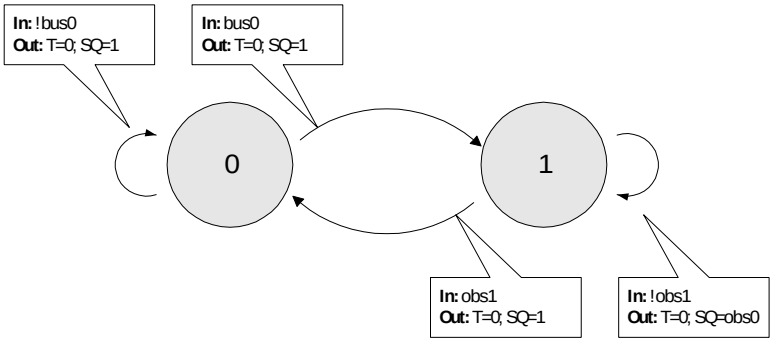
Setting up a sequencer requires the following steps:

1. Building a state diagram.

   A sequence consists of states. The sequencer switches between these states as defined by transition conditions. A state diagram is used to design the sequence.

   State diagrams show the transition conditions and the actions to be performed upon transition (output conditions).

   Example:



2. Programming the pattern terms.

   This is described in *"Programming Pattern Terms" on page 94*.

3. Setting up and programming the sequencer description table.

   The sequencer description table holds the transients. The transients are programmed using C function calls (or CLI commands). The sequencer description table can contain up to 128 transients.

   The state diagram can easily be translated into a sequencer description table. Each transition (arrow) in the diagram requires a transient (a row in the table). Each transient holds the following properties:

   – State

     State to which the transient is assigned (start of the arrow).

   – Next state

     State to which the sequencer should change if the transition condition occurs (end of the arrow).

   – Transition condition

     If this condition is true, the sequencer switches to the "next state".

   – Feedback counters enable condition

     Output conditions controlling the count operation of the feedback counters (not used in this example).

– Feedback counters preload conditions

  Output conditions to set the feedback counters to its preload value (not used in this example).

The trace memory trigger sequencer requires in particular:

– Trigger condition

  Output condition controlling the trigger signal. The trigger signal will only be set if this condition is true and if the transient is active.

– Storage qualifier condition

  Output condition controlling data sampling (storage qualifier). If this condition is true for a trace data line, this line will be stored to trace memory. Otherwise, timestamp information will be stored at the end of the gap (in normal gap mode).

Example:

The following table shows the sequencer description table from the figure.

Transient No.	Current State	Next State	Transition Condition	Trigger Condition	Storage Qualifier
0	0	0	!bus0	0	0
1	0	1	bus0	1	1
2	1	1	!obs0	-	1
3	1	0	obs0	-	1

The sequencer starts in state 0. It observes the transition conditions of the current state and performs the actions as defined for an active transition. If no transition condition is true, the sequencer remains in the current state and no action is taken.

NOTE When programming the sequencer description table, note the following behavior of the feedback counter:

– Clock n: The sequencer instructs the counter to decrement.

– Clock n+1: The counter decrements to terminal count.

– Clock n+2: The terminal count input to sequencer is asserted.

NOTE If the preload condition occurs simultaneously with an increment/decrement condition, the counter amount will be replaced by the preload value, but not incremented or decremented (the preload condition has priority over the count enables).

How to Program the Trigger Sequencer

Programming Steps

Programming the sequencer requires the following steps:

1. Initialize the entire trace memory trigger sequencer to a single-state machine with *BestXTrigDefaultSet*.

   This clears the memory.

2. Set the preload values of the feedback counters A and B.

   Each sequencer is equipped with two preloadable feedback counters. They can be decremented or loaded, enabling you to specify how often a sequence must occur before an output signal is set. Their outputs "tc_fba" and "tc_fbb" (terminal counts) becomes 1 if the counters contain a value of `0xffffffff` (`-1`).

   Use *BestXTrigGenDefaultSet* and *BestXTrigGenSet*.

3. Set all properties in the trigger sequencer description table to default values.

   Use *BestXTrigTranCondDefaultSet*.

4. Set the numeric transition properties "Current State" and "Next State".

NOTE All transition conditions of one state must be mutually exclusive. This means that one and only one transition condition of a state can be true at a time. Otherwise, the software will not accept the table because the table does not uniquely define the sequencer's behavior.

   Use *BestXTrigTranSet*.

5. Set the conditions in the sequencer description table. Conditions can be:

   – transition condition

   – trigger condition

   – storage qualifier condition

   – conditions to decrement and preload the feedback counters

   All conditions are specified as logical expressions.

These expressions can either be set directly to true (1) or false (0), or they can consist of pattern identifiers referring to pattern identifiers (`bus0`, bus1, obs0 … `obs3`, `err0`, `cond0`, `cond1`) and the terminal count (`tc_fba` and `tc_fbb`) of the feedback counters.

For conditions, please refer to "*Conditions Reference*" in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference*.

The programmable pattern terms are used by the sequencer to detect bus state sequences. They compare bus states with programmable conditions (for example, "`b_state==3\h & AD32==b8xxx\h`").

If the programmed condition is true, the sequencer switches to the "Next State". Use *BestXTrigCondSet* for setting conditions.

6 Write the sequencer description table to the sequencer memory.

Use *BestXTrigSeqProg*.

Example for Programming the Trigger Sequencer

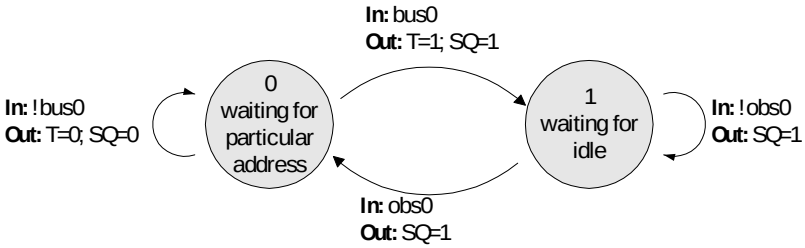
Task The following sequence is to be detected:

- Wait for an address phase that addresses the range 10003000\h … 10003fff\h.
- When the address phase is detected, trigger and store all the transfers.
- Stop storing if an idle cycle occurs.
- Wait for the next access at the specified address.

Solution For this example, the following pattern terms are to be programmed:

- A bus pattern (`bus0`) to detect an address phase that addresses the range `10003000\h` … `10003fff\h`.
- An observer pattern (`obs0`) to detect an active bus state.

The respective state diagram and sequencer description table look as follows:



Transition	Current State	Next State	Transaction Condition	Trigger Condition	Storage Qualifier
0	0	0	!bus0	0	0
1	0	1	bus0	1	1
2	1	1	!obs0	-	1
3	1	0	0bs0	-	1

Implementation

```
#include <xpciapi.h>
#include "SetupUtil.h"

int main(int argc, char* argv[])
{

   BX_TRY_VARS_NO_PROG;

   /* Additional local variable declarations here */

   bx_handletype handle;
   BX_TRY_BEGIN
   {

      BX_TRY(BestXOpen(&handle, BX_PORT_RS232, BX_PORT_COM1));
      BX_TRY(BestXRS232BaudRateSet(handle, BX_BD_57600));

      /* Insert here the C-API calls to set up the block transfer(s)*/

      BX_TRY(BestXTraceStop(handle));

      /* Program the bus pattern. See "bx_signaltype" in
      the Agilent E2929A/B Opt. 320 C-API/PPR Reference.*/

      BX_TRY(BestXPattProg(handle, BX_PATT_BUS0, "AD32==10003xxx\\h
      && addr_phase==1"));
```

```
/* Program the observer pattern. For bstate, the numeric
value for an active bus state is "1". See "bx_signaltype" in
the Agilent E2929A/B Opt. 320 C-API/PPR Reference.*/

BX_TRY(BestXPattProg(handle, BX_PATT_OBS0, "bstate == 1"));

/* Initialize the entire trace memory trigger sequencer */
BX_TRY(BestXTrigDefaultSet( handle ));

/* Set all properties of transitions 0 … 3 of the
description table to defaults */
BX_TRY(BestXTrigTranCondDefaultSet(handle, 0 ));
BX_TRY(BestXTrigTranCondDefaultSet(handle, 1 ));
BX_TRY(BestXTrigTranCondDefaultSet(handle, 2 ));
BX_TRY(BestXTrigTranCondDefaultSet(handle, 3 ));

/* For transition 0:
Set the current state = 0, the next state = 0.
Go to next state, when not "bus0". */

BX_TRY(BestXTrigTranSet(handle, 0, BX_TRIGTRAN_STATE, 0));
BX_TRY(BestXTrigTranSet(handle, 0, BX_TRIGTRAN_NEXTSTATE, 0));
BX_TRY(BestXTrigCondSet(handle, 0, BX_TRIGCOND_X, "!bus0"));

/* Set the trigger condition.*/
BX_TRY(BestXTrigCondSet(handle, 1, BX_TRIGCOND_TRIG, "0"));

/* Set the storage qualifier. */
BX_TRY(BestXTrigCondSet(handle, 0, BX_TRIGCOND_SQ, "0"));

/* For transition 1:
Set the current state = 0, the next state = 1.
Go to next state on "bus0" pattern */

BX_TRY(BestXTrigTranSet(handle, 1, BX_TRIGTRAN_STATE, 0));
BX_TRY(BestXTrigTranSet(handle, 1, BX_TRIGTRAN_NEXTSTATE, 1));
BX_TRY(BestXTrigCondSet(handle, 1, BX_TRIGCOND_X, "bus0"));

BX_TRY(BestXTrigCondSet(handle, 1, BX_TRIGCOND_TRIG, "1"));
BX_TRY(BestXTrigCondSet(handle, 1, BX_TRIGCOND_SQ, "1"));

/* For Transition 2:
Set the current state = 1, the next state = 1.
Go to next state on "!obs0" pattern (!Idle). */

BX_TRY(BestXTrigTranSet(handle, 2, BX_TRIGTRAN_STATE, 1));
BX_TRY(BestXTrigTranSet(handle, 2, BX_TRIGTRAN_NEXTSTATE, 1));
BX_TRY(BestXTrigCondSet(handle, 2, BX_TRIGCOND_X, "!obs0"));

/* Set the storage qualifier */
BX_TRY(BestXTrigCondSet(handle, 2, BX_TRIGCOND_SQ, "1"));
```

```
   /* For Transition 3:
      Set the current state = 1, the next state = 0.
      Go to next state on "obs0" pattern (Idle). */

      BX_TRY(BestXTrigTranSet(handle, 3, BX_TRIGTRAN_STATE, 1));
      BX_TRY(BestXTrigTranSet(handle, 3, BX_TRIGTRAN_NEXTSTATE, 0));
      BX_TRY(BestXTrigCondSet(handle, 3, BX_TRIGCOND_X, "obs0"));
      BX_TRY(BestXTrigCondSet(handle, 3, BX_TRIGCOND_SQ, "1"));

      /* Write the sequencer desciption table contents to the
         testcard */

      BX_TRY(BestXTrigProg(handle));
      BX_TRY(BestXExerciserProg(handle));

      /* Start the test */

      BX_TRY(BestXTraceRun(handle));
      BX_TRY(BestXExerciserRun(handle));


      /* Check if the Analyzer run and was triggered */
      bx_int32 trcstat;
      BX_TRY(BestXStatusRead(handle, BX_STAT_TRC_RUNNING, &trcstat));

      if (trcstat)
      printf("Analyzer running\n");

      BX_TRY(BestXStatusRead(handle, BX_STAT_TRC_TRIGGER, &trcstat));
      if (trcstat)
      {
      printf("Analyzer triggered\n");
      BX_TRY(BestXTraceStop(handle));
      }

      BX_TRY(BestXClose(handle));

   }

   BX_TRY_CATCH
   {
   /* cleanup, if necessary */
   printf("%s\n", BestXErrorStringGet(BX_TRY_RET));
   }

return 0;

}
```

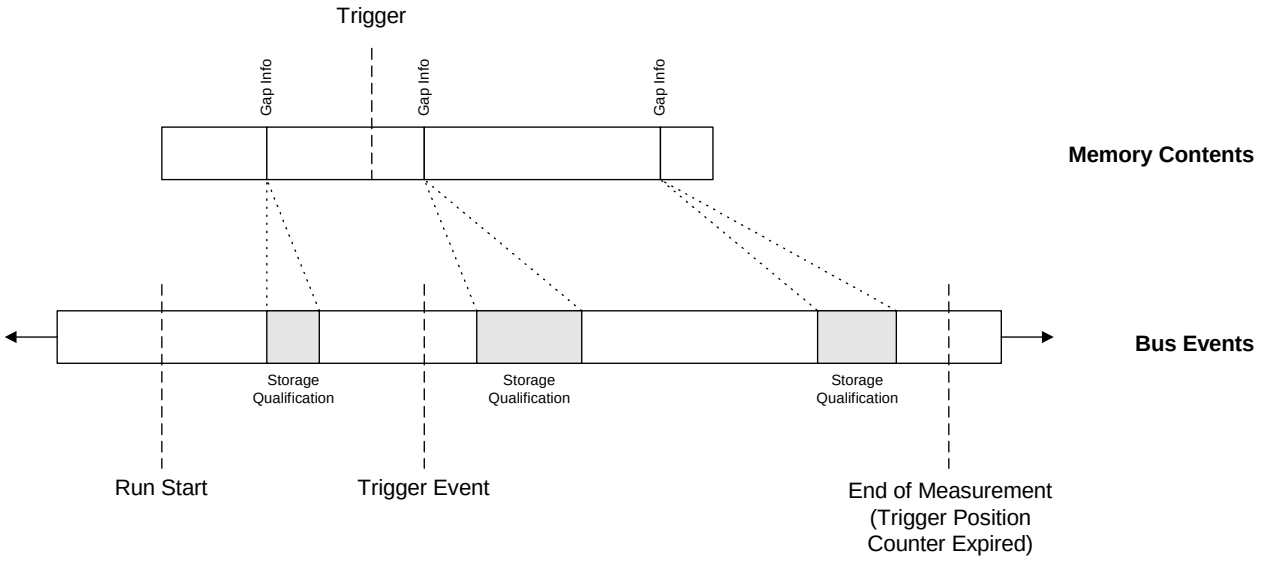
Programming the Trace Memory

The figure below gives an overview of the components of the trace memory.

Filling the Trace Memory

The trace memory is filled depending on storage qualification. In sequencer mode, a **trigger position counter** determines how many states will be sampled into the trace memory after the trigger event occurs. The contents of the trace memory can be controlled by a programmable **storage qualifier** that suppresses undesired states. If one or more lines are filtered, a gap information is stored instead.

The following figure shows how the trace memory is filled.



Before using the trace memory, pattern terms must be defined and the trace memory trigger sequencer must be programmed. See *"Programming Pattern Terms" on page 94* and *"Programming the Trigger Sequencer" on page 96*.

How to Program the Trace Memory

Programming Steps

Programming the trace memory requires the following steps:

1. Program the sequencer and the pattern terms to determine the trigger event. See *"Programming Pattern Terms" on page 94* and *"Programming the Trigger Sequencer" on page 96* for more information.
2. Define the trigger counter preload value that defines how many lines are captured after a trigger event.

   Use *BestXTraceDefault* and/or *WriteBestXTraceWrite*.
3. Run the test by starting the trigger sequencer and the trace memory.

   Use *BestXTraceRun*.

   The test can be monitored by:

   – Polling the *trace status register* with *BestXStatusRead*.

   – Watching the LEDs on the testcard.

   This is particularly useful when the command line interface is used. The LEDs indicate whether trace memory sampling has stopped and whether the trigger has occurred.
4. If you want to stop the run, use *BestXTraceStop*.

   In this case, an artificial trigger point is set. The trace memory contains only samples prior to stoppage (100% pretrigger history).

NOTE The run stops automatically if the memory is full. This can take a lot of time if storage qualifying suppresses a lot of samples.

5. Before you read the data, allocate a data array that is large enough to hold the data.

   For details, see *BestXTraceDataRead* in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference*.
6. Read the trace data from the testcard's trace memory, beginning with the line where the trigger event has occurred, and write it to the host storage with *BestXTraceDataRead*.

   You can also write the complete trace memory content to a file with *BestXTraceDump*.

7 To analyze the data line for certain signals, proceed as follows:

- Determine the position and size of the desired signals within the data line with *BestXTraceBitPosGet*.
- Terminate the connection.
- Displays the results.

Example for Programming the Trace Memory

Task Perform the following task:

- Trigger and store all the transfers in the trace memory, when an address phase that addresses the range 10003000\h … 10003fff\h is detected.
- Stop storing if an idle cycle occurs.
- Read and display the signals stored in the trace memory.

Implementation

```
#include "stdafx.h"
#include <xpciapi.h>
#include "SetupUtil.h"

int main(int argc, char* argv[])
{
   BX_TRY_VARS_NO_PROG;
   /* additional local variable declarations, here */

   bx_handletype handle;
   BX_TRY_BEGIN

   {

      /* Open the connection to the card */

      BX_TRY(BestXOpen(&handle, BX_PORT_RS232, BX_PORT_COM2));

      BX_TRY(BestXRS232BaudRateSet(handle, BX_BD_57600));

      BX_TRY(SetupForTriggerSequencer(handle));
      // You can find this function in SetupUtil.cpp

      BX_TRY(BestXAnalyzerStop(handle));

      /* Insert here the C-API calls to set up the trigger
      sequencer as done in "Example for Programming the Trigger
      Sequencer" on page 118.*/
```

```
/* Now set up block transfers for the testcard. You can use
the following function available in SetupUtil.cpp */
BX_TRY(SetupForBlockTransfer1(handle));

BX_TRY(BestXRIGenSet(handle, BX_RIGEN_NUMBLK, 2));
BX_TRY(BestXRIGenSet(handle, BX_RIGEN_REPEATBLK, 1));
BX_TRY(BestXRIBlockDefaultSet(handle, 0));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_BUSADDR_LO,
                       0x100030fdUL));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_BUSCMD,
                       BX_RIBLK_BUSCMD_MEM_READBLOCK));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_NUMBYTES, 495));
BX_TRY(BestXRIBlockSet(handle, 0, BX_RIBLK_INTADDR, 0));
BX_TRY(BestXRIBlockDefaultSet(handle, 1));
BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_BUSADDR_LO,
                       0x1000ffUL));
BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_BUSCMD,
                       BX_RIBLK_BUSCMD_IO_WRITE));
BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_NUMBYTES, 7));
BX_TRY(BestXRIBlockSet(handle, 1, BX_RIBLK_INTADDR, 0));
BX_TRY(BestXExerciserProg(handle));
BX_TRY(BestXAnalyzerRun(handle));
BX_TRY(BestXTraceRun(handle));
BX_TRY(BestXExerciserRun(handle));
/* Check the status of the Trace memory */
bx_int32 trcstat;
BX_TRY(BestXStatusRead(handle, BX_STAT_TRC_RUNNING,
&trcstat));
if (trcstat)
printf("Analyzer running\n");
```

```
/* Check to see if the trace memory is triggered */
BX_TRY(BestXStatusRead(handle, BX_STAT_TRC_TRIGGER,
                       &trcstat));
if (trcstat)
{
   printf("Analyzer triggered\n");
   BX_TRY(BestXAnalyzerStop(handle));
}
bx_int32 lines;
BX_TRY(BestXStatusRead(handle, BX_STAT_TRC_LINES, &lines));
printf("The number of lines captured is: %d\n",lines);
bx_int32 bytes_per_line;
BX_TRY(BestXTraceBytePerLineGet(handle, &bytes_per_line));
printf("The number of bytes per line is:
        %d\n",bytes_per_line);
bx_int32 total_bytes=lines*bytes_per_line;

/* Now define an array to hold the data from the trace memory
*/
bx_int32 *tptr= (bx_int32 *) malloc(total_bytes);
if (tptr!=0)
{
BX_TRY(BestXTraceDataRead(handle, 0 , lines , tptr));
```

```
/* Define variables for the bit positions and lengths within
the trace memory */

bx_int32 ad32_pos;
bx_int32 ad64_pos;
bx_int32 cbe30_pos;
bx_int32 cbe74_pos;
bx_int32 frame_pos;
bx_int32 irdy_pos;
bx_int32 trdy_pos;
bx_int32 devsel_pos;
bx_int32 stop_pos;

bx_int32 ad32_len;
bx_int32 ad64_len;
bx_int32 cbe30_len;
bx_int32 cbe74_len;
bx_int32 frame_len;
bx_int32 irdy_len;
bx_int32 trdy_len;
bx_int32 devsel_len;
bx_int32 stop_len;

/* Find the bit positions of the various signals within the
trace memory */

BX_TRY(BestXTraceBitPosGet(handle, BX_SIG_AD32, &ad32_pos,
                           &ad32_len));
BX_TRY(BestXTraceBitPosGet(handle, BX_SIG_AD64, &ad64_pos,
                           &ad64_len));
BX_TRY(BestXTraceBitPosGet(handle, BX_SIG_CBE3_0, &cbe30_pos,
                           &cbe30_len));
BX_TRY(BestXTraceBitPosGet(handle, BX_SIG_CBE7_4, &cbe74_pos,
                           &cbe74_len));
BX_TRY(BestXTraceBitPosGet(handle, BX_SIG_FRAME, &frame_pos,
                           &frame_len));
BX_TRY(BestXTraceBitPosGet(handle, BX_SIG_IRDY, &irdy_pos,
                           &irdy_len));
BX_TRY(BestXTraceBitPosGet(handle, BX_SIG_TRDY, &trdy_pos,
                           &trdy_len));
BX_TRY(BestXTraceBitPosGet(handle, BX_SIG_DEVSEL,
                           &devsel_pos, &devsel_len));

BX_TRY(BestXTraceBitPosGet(handle, BX_SIG_STOP, &stop_pos,
                           &stop_len));

/* Find the position of the trigger point in the trace memory
*/

bx_int32 trigpos;

BX_TRY(BestXStatusRead(handle, BX_STAT_TRC_TRIGPOS,
&trigpos));

printf("trigger point at %d\n", trigpos);
```

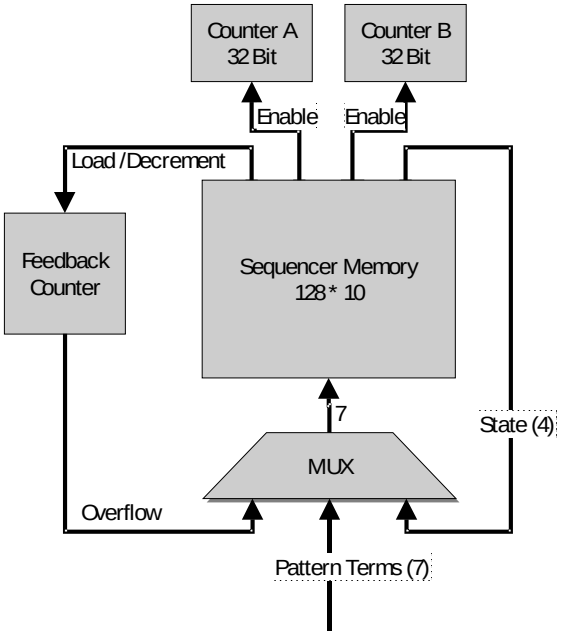
```
      /* Close the connection to the testcard and display the
      results */

      BX_TRY(BestXClose(handle));
      printf("Line #  \tAD64\tAD32\t\tC/BE\tCTRL\n");

      int i;
      int upload_start=0;
      int disp_start=trigpos;

      for (i = 0; i < (lines-upload_start)*bytes_per_line/4;
      i+=(bytes_per_line/4))
      {
         printf("%06d\t%08lx %08lx\t %1lx \t%c%c%c%c%c\n",
         disp_start,
         tptr[i + ad64_pos/32],tptr[i + ad32_pos/32],
         (tptr[i + cbe30_pos/32]>>(cbe30_pos%32)) & ((1<<cbe30_len)-1),
         (((tptr[i + frame_pos/32]>>(frame_pos%32)) & 1) ? ' ' :'F'),
         (((tptr[i + irdy_pos/32]>>(irdy_pos%32)) & 1) ? ' '
            :'I'),
         (((tptr[i + trdy_pos/32]>>(trdy_pos%32)) & 1) ? ' ' : 'T'),
         (((tptr[i + devsel_pos/32]>>(devsel_pos%32)) & 1) ? ' ' :
            'D'),
         (((tptr[i + stop_pos/32]>>(stop_pos%32)) & 1) ? ' ' :
            'S'));

         disp_start++;
      }

      free (tptr);
   }
}

BX_TRY_CATCH

  {
/* cleanup, if necessary */
printf("%s\n", BestXErrorStringGet(BX_TRY_RET));

}
return 0;
}
```

Programming the Performance Sequencer

The testcard features two performance measures that are built up from two 64-bit counters and a sequencer.

The counters of the performance measures are used for real-time performance measurements. They count the occurrences of (freely programmable) events or sequences of events, and thus allow a number of programmable measurements to be registered in real-time.

The following figure shows the hardware components of the performance sequencer.



Performance measurement can be divided into the following steps:

1. Setting up pattern terms.

   For more information, refer to *"Programming Pattern Terms" on page 94*.

2. Programming the sequencer for performance measurement.

   For this purpose, the C-API provides its own function set. See *"How to Program the Performance Sequencer" on page 113*.

3. Running the measurement and viewing the results.

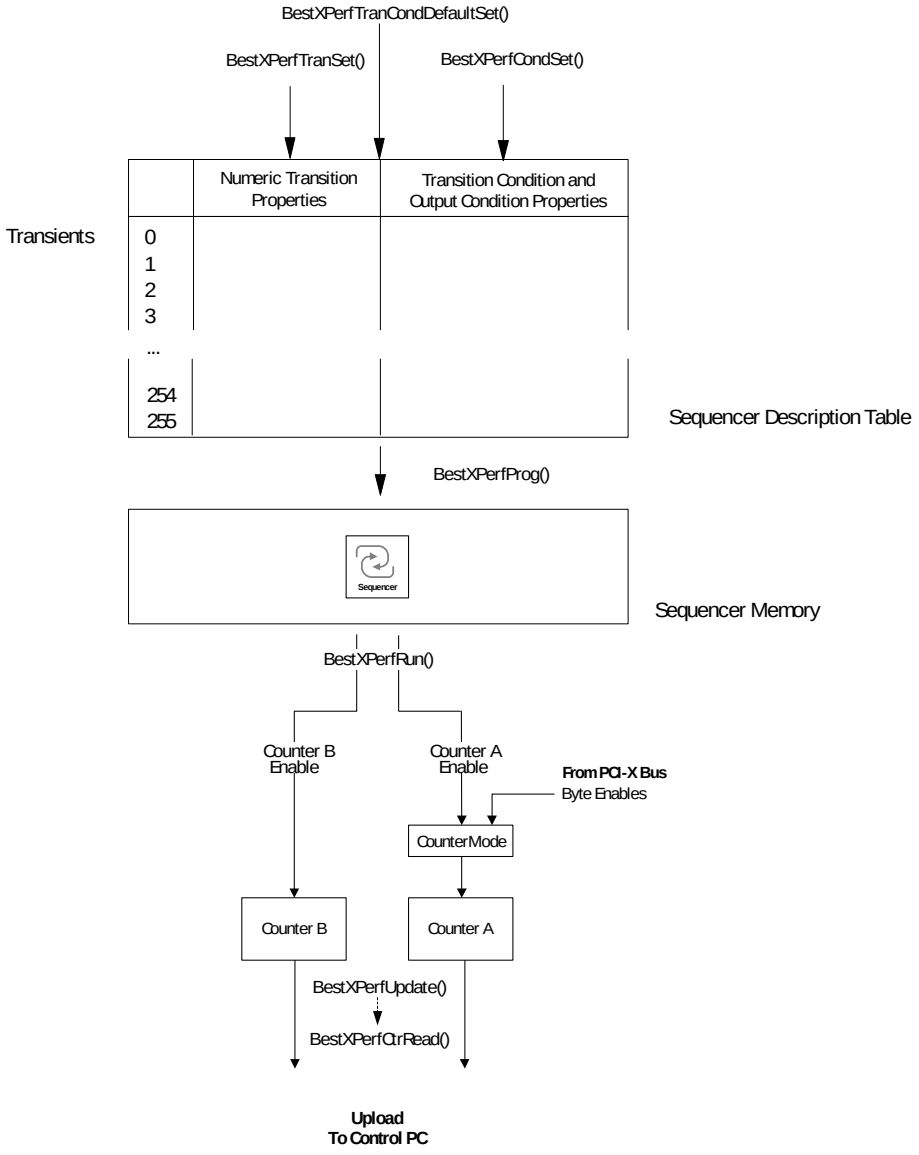
   For this purpose, the measures must be periodically updated, read and the desired values (for example, efficiency) must be computed.

   To compute the desired values, the counter values (reference counter and counters A and B) are needed.

   See *"Example for Programming the Performance Sequencer" on page 115*.

How to Program the Performance Sequencer

The following figure gives an overview of the performance sequencer memory programming model.



Programming Steps

Programming performance measurements requires the following steps:

1. Initialize the entire trace memory trigger sequencer to a single-state machine.

   Use *BestXPerfDefaultSet*.

2. Set the preload value of counter A.

   Use *BestXPerfGenSet*.

   With this function, you can also determine the mode used to increment counter A (increment by one or by the number of transferred bytes).

3. First set all transition and condition properties in the sequencer description table to default values with *BestXPerfTranCondDefaultSet*.

4. Set numeric transition properties “Current State” and “Next State”.

NOTE All transition conditions of one state must be mutually exclusive. This means, that one and only one transition condition of a state can be true at a time. Otherwise, the software will not accept the table because the table does not uniquely define the sequencer’s behavior.

   Use *BestXPerfTranSet*.

5. Set conditions in the performance sequencer description table. Conditions can be:

   – transition condition

   – conditions to increment nominator or denominator counter

   All conditions are specified as logical expressions. These expressions can either be set directly to true (1) or false (0), or they can consist of pattern identifiers.

   If the programmed condition is true, the sequencer switches to the “Next State”.

   Use *BestXPerfCondSet*.

6. Write the sequencer description table to the sequencer memory.

   Use *BestXPerfProg*.

7. To run the measurement, start the counters.

   Use *BestXPerfRun*.

8. To check whether the counters have started, and to check for overflows of each individual counter, read out the performance status register.

   Use *BestXStatusGet*.

9 To compute required data and to view the results, first update the counter values and then read them.

Use *BestXPerfUpdate* and *BestXPerfCtrRead*.

10 You can stop the performance measurement with *BestXPerfStop*.

Example for Programming the Performance Sequencer

Task Display the efficiency and the fraction of non-idle bus states during the exerciser run.

Implementation

```
/* Open the connection to the card */

BX_TRY(BestXOpen(&handle, BX_PORT_RS232, BX_PORT_COM2));
BX_TRY(BestXRS232BaudRateSet(handle, BX_BD_57600));

/* Set up the card for the Trigger Sequencer Example */

BX_TRY(SetupForTriggerSequencer(handle));
BX_TRY(BestXAnalyzerStop(handle));
BX_TRY(BestXPerfGenDefaultSet(handle, BX_PERFMEAS_0));
BX_TRY(BestXPattProg(handle, BX_PATT_OBS0, "bstate == 8"));
BX_TRY(BestXPattProg(handle, BX_PATT_OBS1, "bstate == 1"));
/* For bstate, the numeric value is required. See "bx_signaltype"
in the Agilent E2929A/B Opt. 320 C-API/PPR Reference. */

/* Set the default values for the transient */

BX_TRY(BestXPerfTranCondDefaultSet(handle, BX_PERFMEAS_0, 0));

/* Set the mode for the counters +/

BX_TRY(BestXPerfGenSet(handle, BX_PERFMEAS_0, BX_PERFGEN_CTRAMODE,
                       BX_PERFGEN_CTRAMODE_INCBYTEN));

/* Set up the transient to increment the counters when the
conditions are met */

BX_TRY(BestXPerfTranSet(handle, BX_PERFMEAS_0, 0,
                       BX_PERFTRAN_STATE, 0));

BX_TRY(BestXPerfTranSet(handle, BX_PERFMEAS_0, 0,
                       BX_PERFTRAN_NEXTSTATE, 0));

BX_TRY(BestXPerfCondSet(handle, BX_PERFMEAS_0, 0, BX_PERFCOND_X,
                       "1"));

BX_TRY(BestXPerfCondSet(handle, BX_PERFMEAS_0, 0,
                       BX_PERFCOND_CTRAINC, "obs0"));

BX_TRY(BestXPerfCondSet(handle, BX_PERFMEAS_0, 0,
                       BX_PERFCOND_CTRBINC, "!obs1"));
```

```
/* Run the exerciser infinitely to show some traffic using the
backplane */

BX_TRY(BestXRIGenSet(handle,BX_RIGEN_REPEATBLK,0));

BX_TRY(BestXExerciserProg(handle));

/* Program the testcard and run the performance measurement */

BX_TRY(BestXPerfProg(handle, BX_PERFMEAS_0));
BX_TRY(BestXPerfRun(handle));
BX_TRY(BestXExerciserRun(handle));

bx_int32 counter_a;
bx_int32 counter_b;
bx_int32 ref_counter;

float efficiency;
float nonidle;
while (1)

{

   /* Read off the counters and print to the screen */

   BX_TRY(BestXPerfUpdate(handle));

   BX_TRY(BestXPerfCtrRead(handle, BX_PERFMEAS_0, BX_PERFCTR_A,
                           &counter_a));

   BX_TRY(BestXPerfCtrRead(handle, BX_PERFMEAS_0, BX_PERFCTR_B,
                           &counter_b));

   BX_TRY(BestXPerfCtrRead(handle, BX_PERFMEAS_0, BX_PERFCTR_REF,
                           &ref_counter));

   nonidle= ((float)counter_b / (float)ref_counter) *100;
   efficiency= ((float)counter_a / ((float)counter_b * 4)) * 100;

   printf("Bus Non-idle: %2.2f%% Efficiency: %2.2f%%\r", nonidle,
   efficiency);

}
```

Programming Protocol Permutator and Randomizer Properties

The following sections describe the PCI-X Protocol Permutation and Randomization software and show how to use it.

Background Information

You can find background information in:

- *"Generating Permutations" on page 123* gives basic information about permutations supported with the PPR software.
- *"How to Write a Test Program" on page 127* introduces the steps required for setting up a test program.
- *"Example Test Design" on page 128* shows a typical scenario to be tested. This example is used in all further sections.
- *"Preparing for PPR Programming" on page 131* gives detailed information on the first steps for setting up a test program.

Programming Permutations

You can find information about permutation programming in:

- *"Programming Requester-Initiator Block Permutations" on page 134* gives detailed information on programming and generating requester-initiator block permutations.
- *"Programming RI Behavior Permutations" on page 142* gives detailed information on programming and generating requester-initiator behavior permutations.
- *"Programming CT Behavior Permutations" on page 147* gives detailed information on programming and generating completer-target behavior permutations.
- *"Programming CI Behavior Permutations" on page 150* gives detailed information on programming and generating completer-initiator behavior permutations.
- *"Programming RT Behavior Permutations" on page 153* gives detailed information on programming and generating requester-target behavior permutations.

Running and Analyzing Tests

You can find information about how to run tests and how to set up and analyze test reports in:

- *"Generating PPR Reports" on page 155* gives information on the contents of a PPR report and shows how to program it.
- *"Running a PPR Test" on page 157* shows the required programming steps for running a test.
- *"Analyzing the Report" on page 159* describes all information generated in a PPR Report.
- *"Further Options and Possibilities" on page 172* shows how to optimize the testing time and informs about byte enable variations and uncovered permutations.
- *"Report Listing" on page 174* shows the complete report for the example specified in *"Example Test Design" on page 128*.

Introduction

Developing computer systems requires a lot of different tasks and therefore involves a lot of people. This section outlines the process of computer system development and some roles of those who are involved in it. It shows the benefits of PCI-X Protocol Permutation and Randomization software for each of them.

Computer system development requires the following steps:

- **Device bring-up and debugging**

  The development process starts with the bring-up and debugging phase. In this phase, the devices (add-in testcards, motherboard, and so forth) of a computer system are developed independently by testcard or chipset manufacturers. This phase includes electrical and PCI-X signal integrity tests and finishes with a **functional test phase** at the PCI-X protocol level.

NOTE Corner cases are exhaustive, complicated, and/or uncommon usage of PCI-X protocol elements, thereby indicating system limitations.

This test phase requires a well controllable (but artificial) testing environment. The devices are examined to see whether their protocol level behavior is as expected. The devices are tested on corner cases, whereby coverage of the test cases is well known. The tests are mainly performed by developers of research and development (R&D) departments.

- **System integration**

  After passing these tests, system integrators assemble systems from those testcards. The functionality of the testcards is tested in a **functional test phase**.

  The PCI-X bus is the focus of these examinations, because it connects the motherboard to the peripheral devices within a computer system. Functional tests expose the PCI-X interfaces of devices and motherboard to PCI-X traffic.

  The test checks whether the PCI-X devices of the computer system work as expected. One device after the other is examined, until each of them is exposed to certain functional tests. The tests consider their PCI-X compatibility and again the PCI-X behavior in corner cases at the protocol level.

- **System quality assuranc**e

  In the last phase, the system is exposed to a **system assurance test**. In this phase, it is tested whether all parts of the system cooperate.

  Unlike a functional test, a system assurance test requires a realistic testing scenario. All components must transfer traffic simultaneously. The test result shows whether the system crashes under this stress.

  For system assurance tests, stress tests and performance analysis are performed to find system bottlenecks.

NOTE Testing peripheral devices (such as graphic testcards, SCSI testcards, and LAN testcards) may cause some additional effort (for adapting device drivers or developing test software).

The PCI-X Protocol Permutation and Randomization software provides functional tests for systems and devices at the PCI-X protocol level and system assurance tests.

When testing devices, mainly memory controlling mechanisms can be tested by focusing on host bridges and PCI/PCI-X-to-PCI/PCI-X-bridges.

Contributions of the PCI-X PPR Software

Increased coverage

The PPR software permutates all desired protocol variations within constraints set by the user and by the available exerciser. Because the exerciser can exercise millions of protocol variations (without reprogramming) that are difficult or too time consuming to generate by other means, the test coverage is increased tremendously.

Increased confidence

Unlike in a real system, protocol variations are deterministic and reproducible. After a known number of data transfers, one can be sure that the DUT has been exposed to all desired protocol permutations. A printable report shows which protocol attributes will have been permutated completely against which other protocol attributes as a function of the number of data transfers.

Reduced test time

Because the PCI-X exerciser can be programmed to change its protocol behavior from one data transfer to the next with no CPU overhead in between, the test efficiency approaches 100 %, once the exerciser has been programmed correspondingly. Assuming that validation tests are implemented as a series of nested loops, the same exerciser setup can be used over and over again and thus the setup overhead can be neglected. Overall, this reduces the CPU overhead tremendously.

Reduced development time

The user only needs to set the permutation constraints; the PPR software takes care of the permutation.

Operation Principles

The basic idea is to execute a user-defined block transfer with as many protocol variations as possible, without changing the intention of the block transfer.

The PPR software uses the ability of the PCI-X Exerciser to exercise a series of programmed protocol variations both as a requester and as a completer.

In this section, the particular role of the PCI-X Protocol Permutation and Randomization software is explained. It shows how data transfer is controlled by the Exerciser and Analyzer.

Programmable Memories

The test cases require systematically varying transfer parameters (commands, bytecounts, byte enables). These parameters are controlled by the Exerciser and Analyzer. The information on how to control the parameters is held in programmable memories on the testcard:

- The **requester-initiator block transfer memory** holds control information on how blocks are to be transferred when the Exerciser and Analyzer is used as requester-initiator device (for example, start address alignment, byte enables, bus command).
- The **requester-initiator behavior memory** holds control information on requester-initiator behaviors for each transaction (for example, byte count, address steps, clock delay).
- The **completer-target behavior memory** is used when the testcard is used as completer-target device and holds control information on completer-target behaviors for each transaction (for example, decode speed, initial target response).
- The **completer-initiator behavior memory** is used after a completer-target has given a split response. It holds control information on how split completion transactions are initiated (for example, address steps, clock delay).
- The **requester-target behavior memory** is used when the testcard has to decode split transactions. It holds control information on requester-target behaviors for each transaction (for example, decode speed, initial target response).

The PCI-X Protocol Permutation and Randomization software programs these memories.

NOTE For more information on the memories, refer to *"Programming the Exerciser" on page 23*.

Performing Parameter Permutations

Only the permutation constraints of behaviors and block parameters need to be set, then the permutation and randomizing algorithm first **calculates** whether all possible parameter combinations can be covered and estimates the testing time. The results of the calculation can be written into a **report**. If the algorithm calculated that not all necessary combinations can be covered, it can still be determined which combinations can be performed and which cannot.

The PCI-X Protocol Permutation and Randomization software ensures that the device under test is exposed to all defined protocol variations, thus, PCI-X Protocol Permutation and Randomization software determines the course of the test.

The calculation can be repeated with varying parameters, until the results of the calculation of the PCI-X Protocol Permutation and Randomization software meet your testing requirements. Then the PCI-X Protocol Permutation and Randomization software generates all memory contents, which you can program to the Exerciser using the usual exerciser functions (i.e. BestXExerciserProg).

Start test execution by calling the **exerciser's run functions.** Errors that occur during the test (protocol errors, bus or device hang) can later be analyzed using Exerciser and Analyzer's **analyzer functions**.

Generating Permutations

NOTE Generating permutations using the PCI-X Protocol Permutation and Randomization software takes place in the same way as for PCI.

However, there is no need to understand the permutation algorithms in detail. The software calculates permutations and coverage automatically and shows the results in a report.

Basic Terms

The goal of **permutations** is to combine **values** of different **parameters** (variation parameters) or variables.

Example:

In the following simplified example, 3 different parameters are considered: parameter A, B and C. Each of them holds a **value list**:

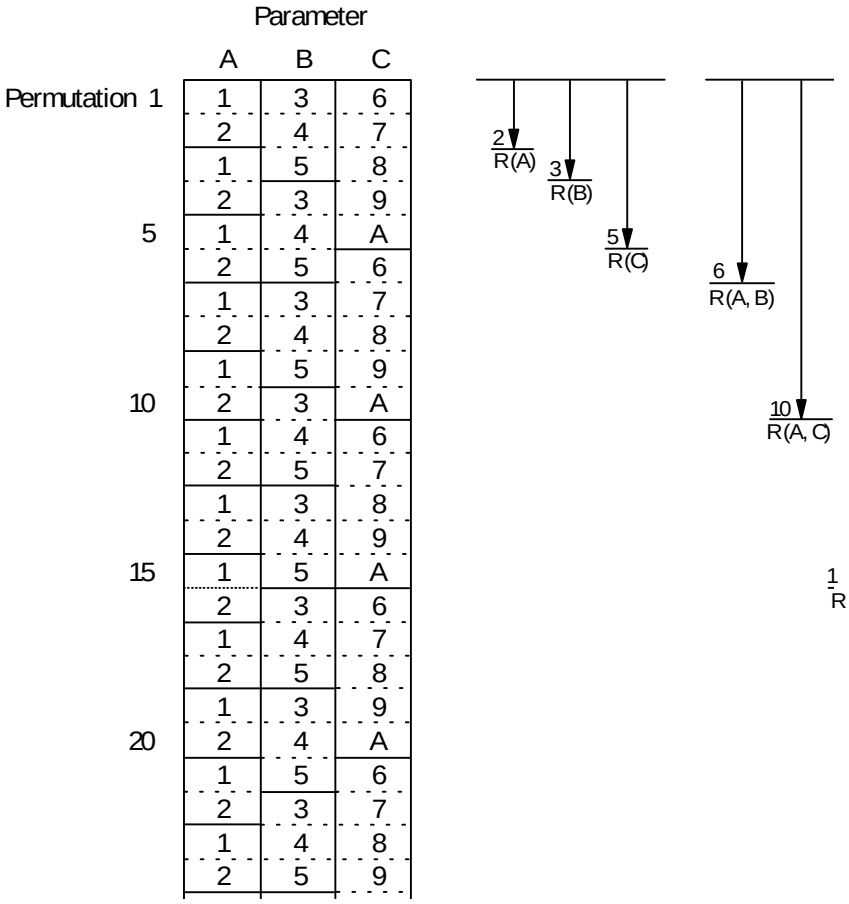
- Parameter A can take the following 2 values: 1 and 2.
- Parameter B can take the following 3 values: 3, 4 and 5.
- Parameter C can take the following 5 values: 6, 7, 8, 9 and A.

Different strategies can be pursued to combine each value of a parameter with all values of the other parameters at least once.

The PCI-X Protocol Permutator and Randomizer software proceeds as follows: it simultaneously works through the value lists of the parameters. With each step—that is each permutation—the next value in the list is combined with the next values in the other lists. Each combination is called a **tuple**.

Permutation Table

Referring to the example, a permutation table would be generated as shown in the following figure. This figure also shows the repetition lengths.



Permutation	A	B	C
1	1	3	6
	2	4	7
	1	5	8
	2	3	9
5	1	4	A
	2	5	6
	1	3	7
	2	4	8
	1	5	9
10	2	3	A
	1	4	6
	2	5	7
	1	3	8
	2	4	9
15	1	5	A
	2	3	6
	1	4	7
	2	5	8
	1	3	9
20	2	4	A
	1	5	6
	2	3	7
	1	4	8
	2	5	9

The software starts with the following permutations:

- It builds the first tuple (tuple 1) from the first values of each list: 1, 3, and 6.
- In the next step, it builds tuple 2 from the second values of each list: 2, 4, and 7.
- In the third step, the list of parameter A has already been worked through. In this case, the software will start again at the beginning of that list building tuples with the remaining values of the other lists. In the example, tuple 3 is built of 1, 5, and 8.

The software proceeds in this way until each value of each parameter is combined with all values of the other parameters, and thus all combinations are covered.

This is the case when the tuples begin to repeat. In the figure, this can be easily seen with the tuples 1 and 7, considering only parameters A and B. After each 6 permutations, the tuple sequence of parameters A and B is repeated.

Repetition Length and Coverage

This number of permutations has therefore been named *repetition length*, written as "R(A, B)".

The repetition length can also be specified for each parameter and is equivalent to the number of values in its value list:

- R(A) is 2
- R(B) is 3
- R(C) is 5

According to the above values, R(A, B) = 6. This equals the product of the repetition lengths of both parameters A and B, namely 2 and 3. The repetition length of the other possible pairs can be calculated in the same way:

- R(A, C) = 2 × 5 = 10
- R(B, C) = 3 × 5 = 15

As can be seen on the previous figure, the tuples built by A and C repeat every 10 permutations, and B and C every 15 permutations.

The repetition length over all parameters is calculated by multiplying the repetition lengths of the particular parameters:

R(A, B, C) = 2 × 3 × 5 = 30.

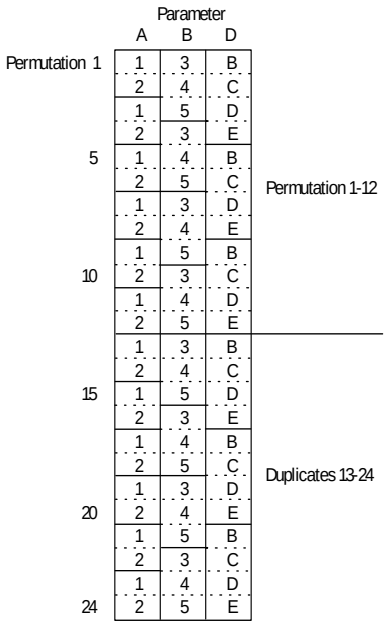
This is represented by the *"Permutation Table" on page 124*. The 31st tuple would again be the same as tuple 1, the 32nd tuple as tuple 2, and so on. This means, that all possible combinations of the values of A, B and C are covered after 30 permutations (*coverage*=30).

Unoccupied Prime Number

Now a new case will be considered: instead of parameter C, a parameter D with the possible values B, C, D, and E should be permutated against parameters A and B. Based on the considerations above, the repetition lengths are calculated as follows:

- R(D) = 4
- R(A, B) = 2 × 3 = 6 (as above)
- R(A, D) = 2 × 4 = 8
- R(B, D) = 3 × 4 = 12
- R(A, B, D) = 2 × 3 × 4 = 24

The following figure, however, shows that this does not work out: the tuples already start to repeat after 12 permutations, although an overall repetition length of 24 was calculated.



	A	B	D
Permutation 1	1	3	B
	2	4	C
	1	5	D
	2	3	E
5	1	4	B
	2	5	C
	1	3	D
	2	4	E
	1	5	B
10	2	3	C
	1	4	D
	2	5	E
	1	3	B
	2	4	C
15	1	5	D
	2	3	E
	1	4	B
	2	5	C
	1	3	D
20	2	4	E
	1	5	B
	2	3	C
	1	4	D
24	2	5	E

Permutation 1-12

Duplicates 13-24

Furthermore, some values are not combined with all other values: for example, there is no tuple containing “1” and “C”, and “2” is never combined with “D”. The reason is that the repetition lengths of parameters A and D have a common factor (2).

To avoid this, the repetition lengths of all involved parameters must not have common factors. The software inserts values into the value lists until the next prime number is reached.

Furthermore, no two parameters may share one prime number as repetition length. For this reason, the PPR software inserts values until the next unoccupied prime number is reached.

In the list of parameter D, in the example, one value would have to be inserted. The list would then hold 5 values, which is the next unoccupied prime number greater than 4.

NOTE The software would insert the first value of the list again. If more than one value had to be inserted, the software would proceed in the order of the values in the list.

These are the basics necessary to understand the meaning of repetition length and coverage and to understand why the PPR software inserts values into the value lists until the repetition lengths are *unoccupied prime numbers*.

How to Write a Test Program

This section gives an overview of how a test session may be built using the PCI-X Protocol Permutator and Randomizer software.

Programming Steps

Programming a test session requires the following steps:

1. Set up the program header.

   The program header contains includes, declarations, and an error handling macro.

   How to program the error handling macro is described in *"Exception Handling" on page 15*.

2. Initialize the software.

   This is done by setting generic properties. See *"Preparing for PPR Programming" on page 131*.

3. Programming permutations

   – Program requester-initiator block permutations.

   See *"Programming Requester-Initiator Block Permutations" on page 134*.

   – Set up requester-intitiator behavior permutations.

   See *"Programming RI Behavior Permutations" on page 142*.

   – Set up completer-target behavior permutations.

   See *"Programming CT Behavior Permutations" on page 147*.

   – Set up completer-initiator behavior permutations.

   See *"Programming CI Behavior Permutations" on page 150*.

   – Set up requester-target behavior permutations.

   See *"Programming RT Behavior Permutations" on page 153*.

4. Set up the report properties.

   For setting up the properties and printing the report, see *"Generating PPR Reports" on page 155*

5 Run the test.

See *"Running a PPR Test" on page 157*.

6 Close up the program.

Free the allocated memory for the PCI-X Protocol Permutation and Randomization software and terminate the Agilent E2920 software. See *"Preparing for PPR Programming" on page 131*.

A complete reference of the available functions can be found in *Agilent E2929A/B Opt. 320 C-API/PPR Reference* or *Agilent E2922A/B Opt. 320 C-API/PPR Reference*.

Example Test Design

To illustrate the basic concepts of the Permutator and Randomizer, an example test is provided. This test is designed to determine whether a compound block can be correctly transferred using various protocol variations.

For this test, it is assumed that:

- The compound block is found in the exerciser's internal block transfer memory at line 0.
- A 1MB-block is to be transferred from the system memory to the exerciser, beginning with the starting address `0x20000000`.
- The system is assumed to provide a 64-bit PCI-X bus.
- Gaps should be filled with fill blocks.

During the transfer, the following protocol variations should occur:

Variations	Variation Parameter	Allowed Values
Requester-Initiator Block Variations	address alignments	0, 1, 2, 3, 4, 5, 6, 7
	byte enables	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
	bus commands	BXPPR_RIBLK_BUSCMD_MEM_READDWORD, BXPPR_RIBLK_BUSCMD_MEM_READBLOCK, BXPPR_RIBLK_BUSCMD_MEM_WRITE, BXPPR_RIBLK_BUSCMD_MEM_WRITEBLOCK
	no snoop	0, 1
	number of bytes	1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128, 256, 512, 1024, 4096
Requester-Initiator Behavior Variations	byte count	33, 64, 72, 128, 4096
	64-bit transfer request	0, 1
	Queue	BX_RIBEH_QUEUE_A, BX_RIBEH_QUEUE_B
	Disconnect	1, 2, 3, 4, 5, 6, 7
	Delay	100, 200, 300

Permutations to be Covered

Different testing areas must be covered during the example test:

- Block variation permutations

  The following permutations of block variation parameters (address alignments, byte enables, bus commands, no snoop, number of bytes) are required:

  – The address alignments 0 … 7 must occur.

  – Byte enables are only valid for DWord commands and the memory write command. Therefore transfers must be executed with 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15 byte enables in the data phase.

  – Transfers must be executed with and without snoop.

  – Transfers must be executed with a block length of 1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128, 256, 512, 1024, and 4096 bytes.

  – Transfers must be executed with memory read DWord and memory read block. Memory write and memory write block transfers cannot be executed, because the programmed direction is *read*.

- Requester-initiator behavior permutations

  The following permutations of requester-initiator behaviors (byte count, 64-bit transfer request, queue, disconnect, delay) are required:

  – Sequences with 33, 64, 72, 128, and 4096 byte counts must be generated.

  – Sequences with and without 64-bit data transfer request must be generated.

  – Sequences must be generated from queue A and queue B.

  – Sequences must be disconnected at every n-th allowable disconnect boundary, where n is 1 … 7.

  – Sequences must be generated after a clock delay of 100, 200, and 300 clock cycles.

- Each block variation permutation must meet each requester-initiator behavior permutation at least once.

Resource Constraints

Resource contraints are determined by the resources of the PCI-X Exerciser and Analyzer testcard available at the moment the test is run. For the example, the following is assumed:

- The blocks contained in the block permutation table must be arranged to fit into the compound block. For the example test, the compound blocksize (CBS) is assumed to be 1MB.
- The block transfer memory can hold a maximum of 256 entries.

  This means that less than 256 blocks may be allocated. Because in this test example a much larger permutation set is created, the test run must be iterated to cover all different combinations.

NOTE The following sections describe how to program the desired permutations and how to get the test results.

The complete implementation of this example can be found under *"Code Listing" on page 185*.

Preparing for PPR Programming

Before the PCI-X PPR software can be used, the testcard and its connections have to be initialized. See *"Synchronizing the Environment" on page 189*.

After the Exerciser and Analyzer software has been initialized, the PPR software must be initialized and generic properties, such as bus speed, bus width, the permutation algorithm and/or the testing level can be set.

Testing Level

Generally, testing can be performed on the protocol level, or on the data level.

- Protocol level testing

  The actual data transferred is irrelevant, the focus is on covering as many protocol variations as allowed. This includes the generation of error conditions.

- Data level testing

  The focus lies on reliable data transfer, so that some variations that indicate faulty behavior can be skipped.

Available Algorithms

The software provides the following algorithms:

- RANDOM

  The randomizing algorithm picks commands from the list at random without eliminating duplicate tuples. Complete coverage can therefore never be guaranteed.

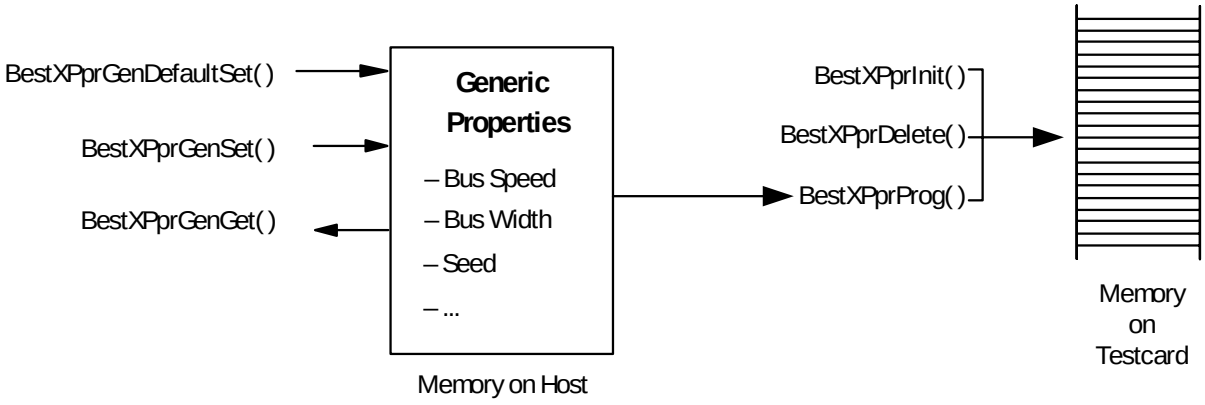
- PERM

  The permutating algorithm picks one command after the other from the list and combines it with the other parameters, regardless of whether the command is suitable or not.

For all available generic properties, refer to "*bx_ribehtype*" in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference* or *Agilent E2922A/B Opt. 320 C-API/PPR Reference*.

How to Prepare for PPR Programming

The following figure shows the generic setup functions used to initialize and deinitialize the PCI-X Protocol Permutation and Randomization software and for getting and setting general properties.



Programming Steps To set up the test program, the following steps are required:

1. Initialize the Exerciser and Analyzer testcard.

   See *"Getting Started" on page 17*.

2. Initialize the PPR software by setting all properties of this software to default values.

   Use *BestXPprInit*.

3. Recommended: Set all generic properties to default values. Use *BestXPprGenDefaultSet*.

4. Set general properties, such as PCI-X bus speed and bus width, the expected number of clocks per data transfer and a random seed.

   Use *BestXPprGenSet*. Using this function also allows you to define which part of the exerciser will be programmed to the testcard. See *BestXPprProg*.

   To read the settings, use *BestXPprGenGet*.

5. Write all current settings to the testcard with *BestXPprProg*.

NOTE Before you can use this function, the Exerciser must be stopped.

6. At the end of the test, free all memory allocated by the software.

   Use *BestXPprDelete*.

Example for Preparing for PPR Programming

Task Program the following generic properties:

- PCI-X bus width in bits
- PCI-X bus speed in Hz
- Expect 5 clocks per data transfer
- Perform testing on data level
- Use the permutation algorithm to permutate blocks and behaviors
- Perform testing with ADB limitation

Implementation

```
/* Initialize the PPR software */
BX_TRY(BestXPprInit(handle));
/* Set all PPR generic properties to their defaults */
BX_TRY(BestXPprGenDefaultSet(handle));
/* Set all RI PPR properties to their defaults */
BX_TRY(BestXPprRIDefaultSet(handle));
/* Set the PPR generics */
BX_TRY(BestXStatusRead(handle, BX_STAT_BUSWIDTH, &width));
BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_BUSWIDTH, width));
BX_TRY(BestXStatusRead(handle, BX_STAT_BUSSPEED, &speed));
BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_BUSSPEED, speed));
BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_XFERCLKS, 5)); // THIS IS
                                                    //JUST AN ESTIMATE
BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_ALGORITHM,
                        BXPPR_GEN_ALGORITHM_PERM));
BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_LEVEL,
                        BXPPR_GEN_LEVEL_DATA));
BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_ADBLIMITATION , 1));
```

Programming Requester-Initiator Block Permutations

This section describes how a block transfer is prepared and explains the properties important for the block transfer and the permutations.

Block

A block is a contiguous range in the memory that is to be transferred *with one single command*. This transfer, however, is always initiated by a requester-initiator and may require multiple bursts to complete, due to the requester-initiator intention, completer-target termination, or an intervention of the arbiter.

Compound Block

A compound block consists of several, smaller blocks, which are transferred by the exerciser as a sequence. The sequence is split into transactions according to terminations issued by the requester-initiator or the completer-target.

If the sequence is terminated with a split termination, the completer-initiator completes this sequence.

Block Permutation Properties

Block permutation properties define the intention of the compound block. The following properties can be set:

- Transfer Direction (`BXPPR_RIBLKPERM_DIRECTION`)

  The transfer direction is seen from the requester-initiator side:

  – *write*

  From the Exerciser to system memory

  – *read*

  From system memory to the Exerciser

  – *readcompare*

  System memory contents are compared with the exerciser contents.

  – *writereadcompare*

  Data are transferred from the exerciser to the system memory, read back and compared with the original values.

- Compound Block Size (BXPPR_RIBLKPERM_BLOCKSIZE)

  The compound block size (CBS) specifies the size of the compound block in dwords.

  The permutation algorithm fits the blocks into this compound block according to the required block variation constraints. See *"Block Variation Parameters" on page 136* for constraints.

NOTE It is recommended that the compound block size is set to a power of 2.

- Bus Address (BXPPR_RIBLKPERM_BUSADDRESS_LO, BXPPR_RIBLKPERM_BUSADDRESS_HI)

  The bus address is the starting address in the PCI-X memory range of the system under test to which the compound block will be transferred, or from which it will be read.

WARNING Always allocate the required memory for your test program. If you write directly into system memory (bypassing by the operating system), the system may seriously crash.

- Resource (BXPPR_RIBLKPERM_RESOURCE)

  Defines the internal resource (data memory or data generator) to which the compound block will be transferred, or from which it will be read.

- Internal Address (BXPPR_RIBLKPERM_INTADDR)

  The internal address is the starting address within the internal resource of the testcard (data memory or data generator) to which the compound block will be transferred, or from which it will be read.

NOTE The PCI-X Protocol and Randomizer software does not fill up memory with data. This can be done with the appropriate standard C-API functions. See *"Programming the Data Memory" on page 81*.

- First Permutation Number (BXPPR_RIBLKPERM_FIRSTPERM)

  This value is used to start the permutation algorithm with a certain value. It can be used to continue a permutation if a previous permutation had to be interrupted, for example, because of an overflowing block permutation memory.

- Fill Gaps (BXPPR_RIBLKPERM_FILLGAPS)

  This boolean value determines whether or not gaps between blocks in the compound block are filled after fitting in block permutations. Filling these gaps ensures that the whole compound block will be transferred. However, to fill the gaps, all address alignments and byte

enable values will be used, not just the values specified for these parameters.

- Start Offset (`BXPPR_RIBLKPERM_STARTOFFSET`)

  Defines at which line of the requester-initiator block permutation memory the programming begins.

- Size Limit (`BXPPR_RIBLKPERM_SIZELIMIT`)

  Limits the requester-initiator block permutation memory usage.

- Tuples (`BXPPR_RIBLKPERM_TUPLES`)

  Maximum number of groups that are permutated against each other for calculation of coverage.

Block Variation Parameters

Block variation parameters specify how the compound block is to be intensified by permutated variations of parameters, such as block size or alignment.

These parameters can be constrained to design a test scenario according to the testing requirements. To constrain a parameter, a list of values to be permutated and an algorithm for picking the values from the list can be specified. The algorithm selects values either at random or sequentially. To specify an algorithm, see *"How to Prepare for PPR Programming" on page 132*.

The following block variation parameters are available:

- Bus Commands (`BXPPR_RIBLK_BUSCMD`)

  A list of PCI-X bus commands can be specified. All PCI-X bus commands can be specified, but only those commands that are suitable for the specified transfer direction will be used for variations.

NOTE Because there are no limitations on alignments for any command, the permutation of commands is straightforward. Because the breaking-down of bursts into single transfers for DWord commands is performed in the hardware, no special handling is necessary.

I/O commands and memory commands cannot be mixed.

NOTE Usage of DWord commands limits variation on termination, `REQ64` usage and `RELREQ` permutations.

- Alignment (`BXPPR_RIBLK_ALIGN`)

  A list of alignments to the 128-byte ADB can be specified.

  Permutation on alignments leaves gaps between blocks, which can either be filled automatically or left uncovered. See *"Fill Gaps (BXPPR_RIBLKPERM_FILLGAPS)" on page 135*.

- Byte Enables (BXPPR_RIBLK_BYTEN)

  A list of byte enables can be specified to occur in the data phase of the block transfer.

  Byte enables are only valid for the following commands:

  BX_BUSCMD_MEM_WRITE, BX_BUSCMD_MEM_READDWORD, BX_BUSCMD_IO_READ, and BX_BUSCMD_IO_WRITE.

  Therefore, permutation is only performed if one of these commands is in the commands variation list.

NOTE Alignments and byte count settings may limit byte enables to the valid data. This can introduce byte enables that are not in the variation list or could even lead to a parameter being shipped from the list completely. This is stated in the report.

- Number of Bytes (BXPPR_RIBLK_NUMBYTES)

  Defines the number of bytes for the current block.

- Relaxed order (BXPPR_RIBLK_RELAXORDER)

  Defines if the relaxed ordering bit is shown in the attribute phase.

- No snoop (BXPPR_RIBLK_NOSNOOP)

  Defines whether snoop will be done.

Coverage The software computes whether all blocks required for the permutations fit into the specified compound block. Coverage is achieved if all possible permutations are covered after all blocks in the compound block have been transferred. The result of this computation can be written to a report.

The coverage of the requester-inititiator block permutation depends on the number of variation parameters examined, the PCI-X bus commands used, and the algorithm that selects the parameter combination for each permutation step.

Calculations of Coverage

The following table describes the scheme used by the software to determine the coverage of the variation list.

Direction	Algorithm	
	RAND	**PERM**
READ	No coverage can be guaranteed	Coverage = Repetition length of commands in the list, raised up to the next prime
WRITE		Coverage = Repetition length of **all** commands in the list, raised up to the next prime × Repetition length of the influencing parameters, raised up to the next prime

To compute the coverage information, the software works through the specified block variation parameters, through all of their allowed values, and creates a block with each parameter combination. Illegal combinations are replaced by legal ones. Because these valid combinations may have occurred before, this may produce duplicate combinations. Such duplicates will be skipped automatically.

NOTE The allocated memory must be freed before the Exerciser and Analyzer software is terminated.

See *"How to Prepare for PPR Programming" on page 132*.

After a number of blocks (variation list length N), duplicate blocks would be created. Thus, a block property is covered after N data transfers. For example, two data transfers are required to test a block that consists of two address alignment values.

The number of data transfers needed to guarantee that each block property value is calcutated by multiplying the number of values of each property. For example, to combine 5 address alignments with 3 block sizes, $5 \times 3 = 15$ data transfers (blocks) are required.

The variation list lengths N and repetition lengths R can be queried or can be found in the report.

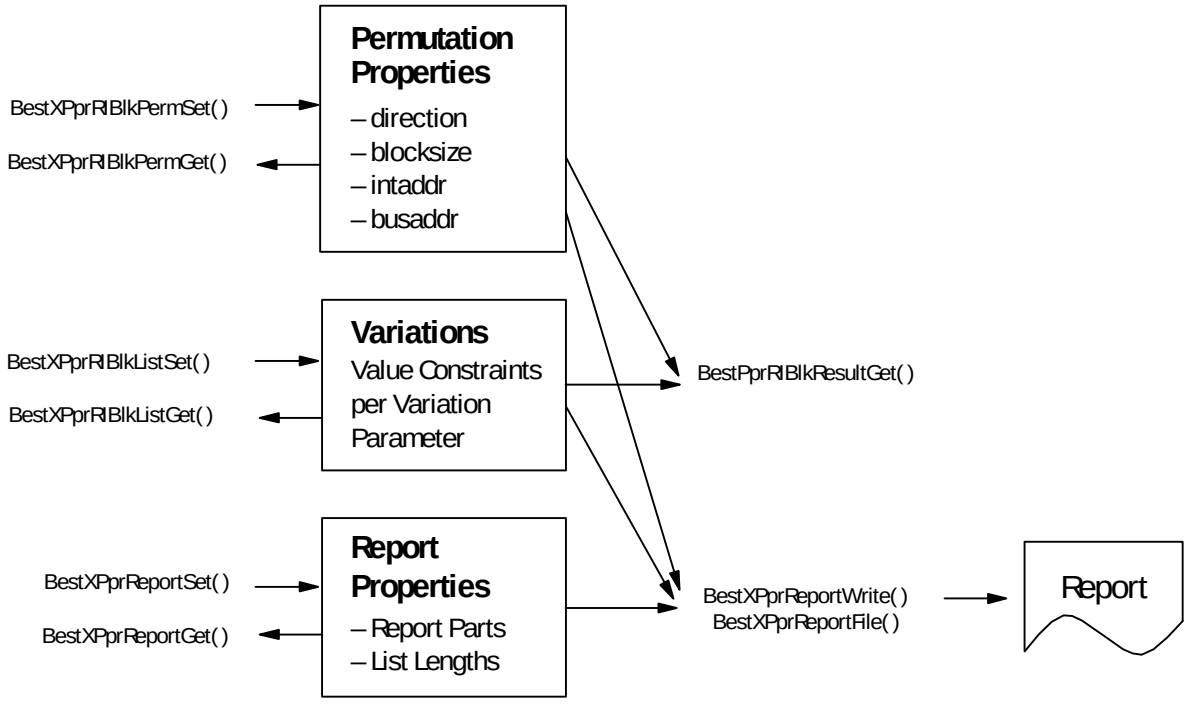
NOTE The calculated test coverage only indicates which protocol permutations are intended to be used. The device under test will be exposed to all permutations, but it cannot be guaranteed that a transfer will take place using each permutation (for example, due to specific device characteristics or malfunctions).

Testing Time The testing time required to execute the compound block on the testcard can be printed to the report. It consists of the compound block size multiplied by the time needed per data transfer.

Contribution	Testing Time
Requester-Initiator Block Permutation Testing Time	CBS × Time-per-data-transfer

How to Program RI Block Permutations

The following figure shows the requester-initiator block and the report functions used to prepare and to perform a requester-initiator block permutation.



Programming Steps

To program requester-initiator block permutations, the following steps are required:

1. Prepare a permutation by setting the block permutation properties (for example, transfer direction, bus address and internal address).

   Use *BestXPprRIBlkPermSet*.

2. Define lists of values for the variations with *BestXPprRIBlkListSet*.

   These lists specify values for the block variation parameters to be permutated according to the testing requirements. Block variation parameters are alignment, block size, values of the C/BE lines in the data phase and block commands.

3. If you want to check whether your test requirements are really suitable, request the test results, such as the actually used block memory size and the number of the last requester-initiator block permutation.

   Use *BestXPprRIBlkResultGet*.

   If the number of permutations is larger than the number of possible block memory entries (256), execution of permutations must be iterated to run all different combinations. See *"Example for Programming RI Block Permutations" on page 141*.

4. Write all PPR settings to the testcard with *BestXPprProg*.

5. The permutation results can be requested from the PCI-X Protocol Permutation and Randomization software. Adjusted properties and permutation results can then be written to a report file.

   Use *BestXPprReportWrite* or *BestXPprReportFile*.

NOTE The contents of the report file can be controlled with *BestXPprReportSet* and *BestXPprReportGet*.

Example for Programming RI Block Permutations

Task For this test, it is assumed that:

- The compound block is found in the exerciser's internal block transfer memory at line 0.
- A 1MB-block is to be transferred from the system memory to the exerciser, beginning with the starting address `0x20000000`.
- The system is assumed to provide a 64-bit PCI-X bus.
- Gaps should be filled with fill blocks.

During the transfer, the following protocol variations should occur:

Variations	Variation Parameter	Allowed Values
Requester-Initiator Block Variations	address alignments	0, 1, 2, 3, 4, 5, 6, 7
	byte enables	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
	bus commands	BXPPR_RIBLK_BUSCMD_MEM_READDWORD, BXPPR_RIBLK_BUSCMD_MEM_READBLOCK, BXPPR_RIBLK_BUSCMD_MEM_WRITE, BXPPR_RIBLK_BUSCMD_MEM_WRITEBLOCK
	no snoop	0, 1
	number of bytes	1, 2, 3, 4, 5, 6, 7, 8, 16, 32, 64, 128, 256, 512, 1024, 4096

Implementation

```
/* Program the block permutation properties */
BX_TRY(BestXPprRIBlkPermSet(handle, BXPPR_RIBLKPERM_DIRECTION,
                            BXPPR_RIBLKPERM_DIRECTION_READ));
BX_TRY(BestXPprRIBlkPermSet(handle, BXPPR_RIBLKPERM_BLOCKSIZE,
                            0x100000)); // a 1M block
BX_TRY(BestXPprRIBlkPermSet(handle, BXPPR_RIBLKPERM_BUSADDR_LO,
                            0x20000000));
BX_TRY(BestXPprRIBlkPermSet(handle, BXPPR_RIBLKPERM_BUSADDR_HI,
                            0x30000000));
BX_TRY(BestXPprRIBlkPermSet(handle, BXPPR_RIBLKPERM_INTADDR,
                            0x0000));
BX_TRY(BestXPprRIBlkPermSet(handle, BXPPR_RIBLKPERM_FILLGAPS,
                            BX_YES)); // this is the default
/* Start filling the Block transfer memory at line 0 of 256 */
BX_TRY(BestXPprRIBlkPermSet(handle, BXPPR_RIBLKPERM_STARTOFFSET,
                            0)); //this is the default
/* Define the permutation lists.  These lists of values will be
permutated against each other */
```

```
/* The permutated BUSCMD values are dependent on
BXPPR_RIBLKPERM_DIRECTION */

BX_TRY(BestXPprRIBlkListSet(handle, BXPPR_RIBLK_BUSCMD, \
                            "BX_RIBLK_BUSCMD_MEM_READDWORD,
                            BX_RIBLK_BUSCMD_MEM_READBLOCK,
                            BX_RIBLK_BUSCMD_MEM_WRITE,
                            BX_RIBLK_BUSCMD_MEM_WRITEBLOCK"));

BX_TRY(BestXPprRIBlkListSet(handle, BXPPR_RIBLK_NOSNOOP, "0,1"));

BX_TRY(BestXPprRIBlkListSet(handle, BXPPR_RIBLK_ALIGN,
                            "0,1,2,3,4,5,6,7"));

BX_TRY(BestXPprRIBlkListSet(handle, BXPPR_RIBLK_BYTEN,
                            "0,1,2,3,4,5,6,7,8,9,
                            10,11,12,13,14,15"));

BX_TRY(BestXPprRIBlkListSet(handle, BXPPR_RIBLK_NUMBYTES,
                            "1,2,3,4,5,6,7,8,16,32,64,
                            128,256,512,1024,4096"));
```

Programming RI Behavior Permutations

The requester-initiator behaviors can be constrained and permutated in the same way as the requester-initiator block variation parameters described above.

Requester-Initiator Behaviors

To achieve more sophisticated randomization opportunities, the requester-initiator behaviors are divided into groups, which are varied against each other. The following tables show which behaviors are assigned to which group:

Group	Behaviors
Group 0	Queue, Steps, Req64
Group 1	Byte Count
Group 2	Disconnect
Group 3	Delay, RelReq

Variation Parameters

A list of values can be specified for every requester-initiator behavior.

Coverage

To get the coverage result, the requester-inititiator behaviors are first permutated against required behaviors within their own group. The resulting repetition length is increased to the next higher unoccupied prime number greater than 2—the prime number 2 is skipped to obtain an odd requester-initiator behavior page size.

NOTE For an explanation of how the permutations are generated, refer to *"Generating Permutations" on page 123*.

Finally, the algorithm computes the number of data transfers required to achieve complete coverage by internally permutating the behavior groups against each other.

The *repetition lengths* per group and the *coverage information* per group and per group combination (tuples) can be found in the report. The algorithm also calculates how many block runs are needed to cover all required combinations, and determines the amount of data to be transferred. Additionally, this information can also be queried using C-API functions.

Testing Time

The testing time is determined by the number of PCI-X data transfers resulting from the number of groups and their lines to be permutated in the requester-initiator behavior memory.

Testing Time for Requester-Initiator Behavior Permutations
Number of data transfers × Time-per-data-transfer

Block vs. Requester-Initiator Behavior Permutation

The requester-initiator block parameters can be permutated against requester-initiator behaviors. Requester-initiator behaviors are permutated through their values when a compound block is executed repeatedly.

After the compound block has been completely executed with each requester-initiator behavior group combination, all permutated block variation properties have been permutated against all permutated requester-initiator behaviors. The algorithm calculates how many block runs are needed to cover all required requester-initiator behaviors and their combinations and the amount of data to be transferred.

If the compound block size has been set to a power of 2, it is ensured that all permutated block variation properties have permutated against all permutated requester-initiator behaviors.

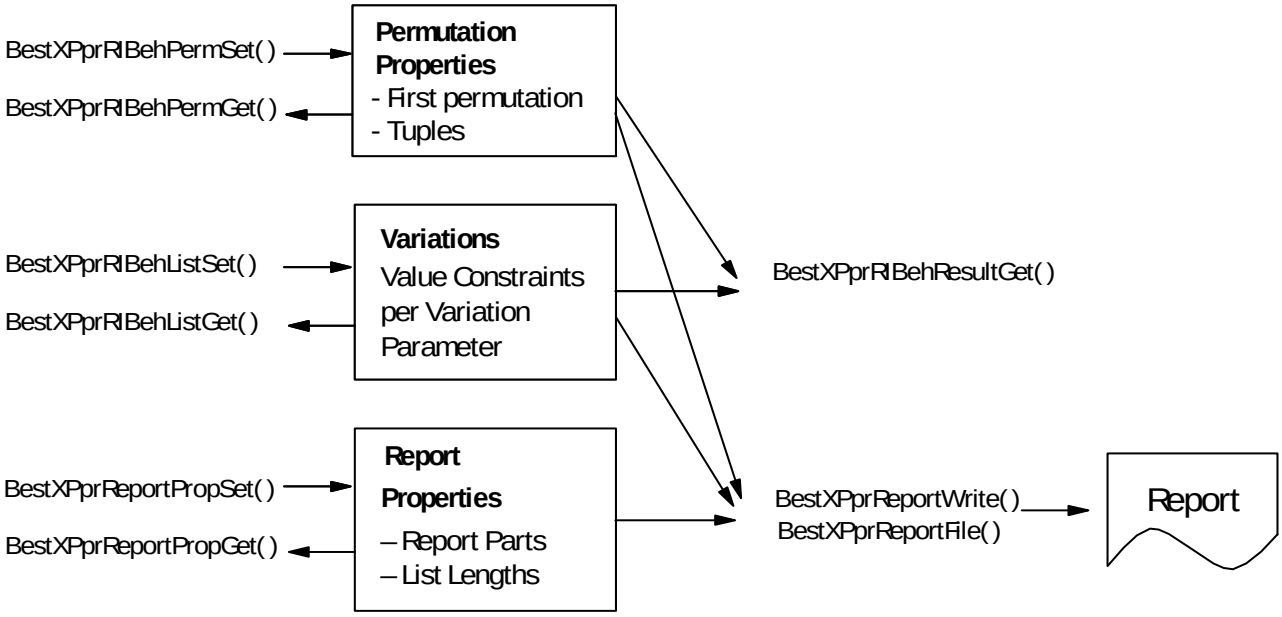
Testing Time

The testing time is determined by the testing time for the requester-initiator block permutations and the number of requester-initiator behavior permutations.

Testing Time
Testing Time for Requester-Initiator Block Permutations × Number of Behavior Permutations

How to Program RI Behavior Permutations

The following figure shows the requester-initiator behavior and report functions used to prepare and to perform the permutation of the requester-initiator behavior.



Programming Steps

Programming requester-initiator behavior permutations requires the following steps:

1 Prepare a permutation by setting the permutation properties according to the resource requirements.

Use *BestXPprRIBehPermSet*.

Requester-initiator behavior permutation properties are:

– the maximum number of groups that are permutated against each other for calculation of coverage

– a starting point for the permutation algorithm

2 Define the lists of values for the variations with *BestXPprRIBehListSet*.

These lists specify values for the requester-initiator behavior properties to be permutated according to the testing requirements. For the behavior properties, refer to "*bx_ribehtype*" in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference* or *Agilent E2922A/B Opt. 320 C-API/PPR Reference*.

3 If you want to check whether your test requirements are really suitable, request the test results or coverage.

Use *BestXPprRIBehResultGet*.

4 Write all PPR settings to the testcard with *BestXPprProg*.

5 The permutation results can be requested from the PCI-X Protocol Permutator and Randomizer software. Adjusted properties and permutation results can then be written to a report file.

Use *BestXPprReportWrite* or *BestXPprReportFile*.

NOTE The contents of the report file can be controlled with *BestXPprReportSet* and *BestXPprReportGet*.

Example for Programming RI Behavior Permutations

Task Program the following requester-initiator behavior variations:

Variations	Variation Parameter	Allowed Values
Requester-Initiator Behavior Variations	byte count	33, 64, 72, 128, 4096
	64-bit transfer request	0, 1
	Queue	BX_RIBEH_QUEUE_A, BX_RIBEH_QUEUE_B
	Disconnect	1, 2, 3, 4, 5, 6, 7
	Delay	100, 200, 300

Implementation

```
BX_TRY(BestXPprRIBehPermDefaultSet(handle));

BX_TRY(BestXPprRIBehListSet(handle, BX_RIBEH_BYTECOUNT, "33, 64,
                                72, 128, 4096"));

BX_TRY(BestXPprRIBehListSet(handle, BX_RIBEH_REQ64, "0,1"));

BX_TRY(BestXPprRIBehListSet(handle, BX_RIBEH_QUEUE,
                            "BX_RIBEH_QUEUE_A, BX_RIBEH_QUEUE_B"));

BX_TRY(BestXPprRIBehListSet(handle, BX_RIBEH_DISCONNECT,
                            "1,2,3,4,5,6,7"));

BX_TRY(BestXPprRIBehListSet(handle, BX_RIBEH_DELAY,
                            "100,200,300"));

BX_TRY(BestXPprRIBehPermSet(handle, BXPPR_BEHPERM_FIRSTPERM, 1));
```

Programming CT Behavior Permutations

The completer-target behaviors can be constrained and permutated in the same way as the requester-initiator block variation parameters described above.

Completer-Target Behaviors

To achieve more sophisticated randomization opportunities, the completer-target behaviors are divided into groups, which are varied against each other. The following tables show which behaviors are assigned to which group:

Group	Behaviors
Group 0	DecSpeed, Ack64, SplitLatency
Group 1	Initital, Latency
Group 2	Subseq, SubseqPhase, SplitEnable

Variation Parameters

A list of values can be specified for every completer-target behavior.

Coverage

To get the coverage result, the completer-target behaviors are first permutated against required behaviors within their own group. The resulting repetition length is increased to the next higher unoccupied prime number greater than 2—the prime number 2 is skipped to obtain an odd completer-target behavior page size.

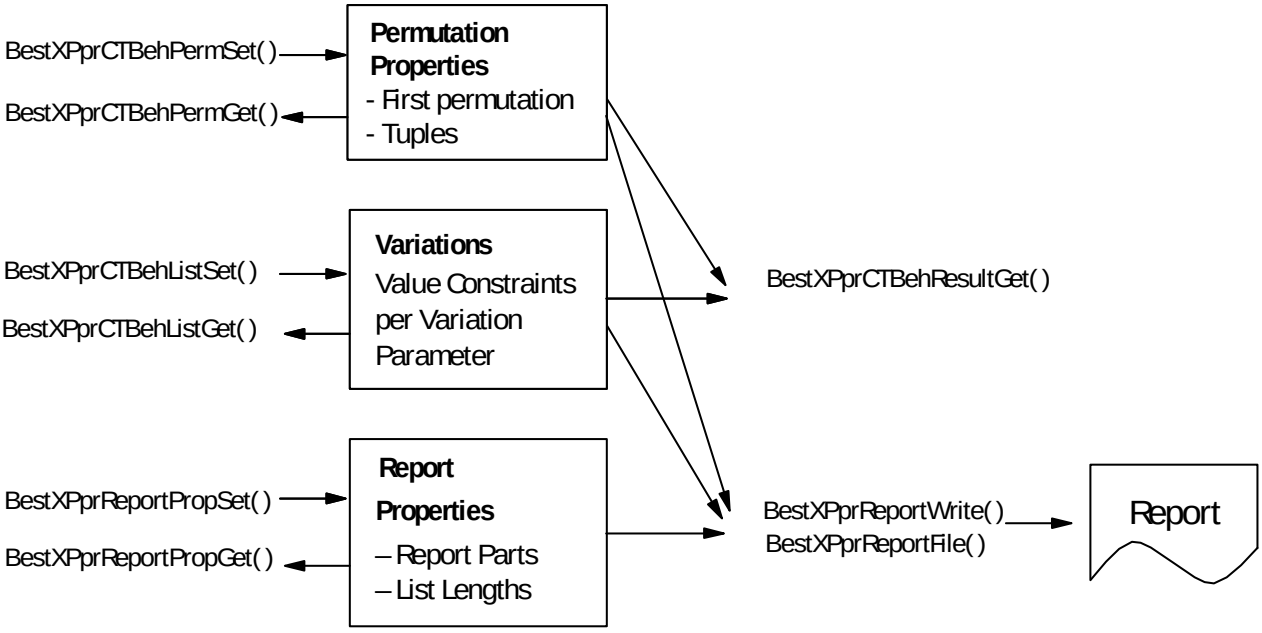
NOTE For an explanation of how the permutations are generated, refer to *"Generating Permutations" on page 123*.

Finally, the algorithm computes the number of data transfers required to achieve complete coverage by internally permutating the behavior groups against each other.

The *repetition lengths* per group and the *coverage information* per group and per group combination (tuples) can be found in the report.

How to Program CT Behavior Permutations

The following figure shows the completer-target behavior and report functions used to prepare and to perform the permutation of the completer-target behaviors.



Programming Steps

To program completer-target behavior permutations, the following steps are required:

1 Prepare a permutation by setting the permutation properties according to the resource requirements.

Use *BestXPprCTBehPermSet*.

Completer-target behavior permutation properties are:

– the maximum number of groups that are permutated against each other for calculation of coverage

– a starting point for the permutation algorithm

2 Define the lists of values for the variations with *BestXPprCTBehListSet*.

These lists specify values for the completer-target behavior properties to be permutated according to the testing requirements. For the behavior properties, refer to "*bx_ctbehtype*" in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference* or *Agilent E2922A/B Opt. 320 C-API/PPR Reference*.

3 If you want to check whether your test requirements are really suitable, request the test results or coverage.

   Use *BestXPprCTBehResultGet*.

4 Write all PPR settings to the testcard with *BestXPprProg*.

5 The permutation results can be requested from the PCI-X Protocol Permutator and Randomizer software. Adjusted properties and permutation results can then be written to a report file.

   Use *BestXPprReportWrite* or *BestXPprReportFile*.

NOTE The contents of the report file can be controlled with *BestXPprReportPropSet* and *BestXPprReportPropGet*.

Example for Programming CT Behavior Permutations

In general, the setup of the C code for programming the completer-target behavior permutations takes place in the same way as described for requester-initiator permutations.

Refer to *"Example for Programming RI Behavior Permutations" on page 146* for programming requester-initiator behavior permutations.

Programming CI Behavior Permutations

The completer-initiator behaviors can be constrained and permutated in the same way as the requester-initiator block variation parameters described above.

Completer-Initiator Behaviors

To achieve more sophisticated randomization opportunities, the completer-initiator behaviors are divided into groups, which are varied against each other. The following tables show which behaviors are assigned to which group:

Group	Behaviors
Group 0	Queue
Group 1	ErrMsg
Group 2	Partition
Group 3	Delay, RelReq
Group 4	Steps, Req64

Variation Parameters

A list of values can be specified for every completer-initiator behavior.

Coverage

To get the coverage result, the completer-initiator behaviors are first permutated against required behaviors within their own group. The resulting repetition length is increased to the next higher unoccupied prime number greater than 2—the prime number 2 is skipped to obtain an odd completer-initiator behavior page size.

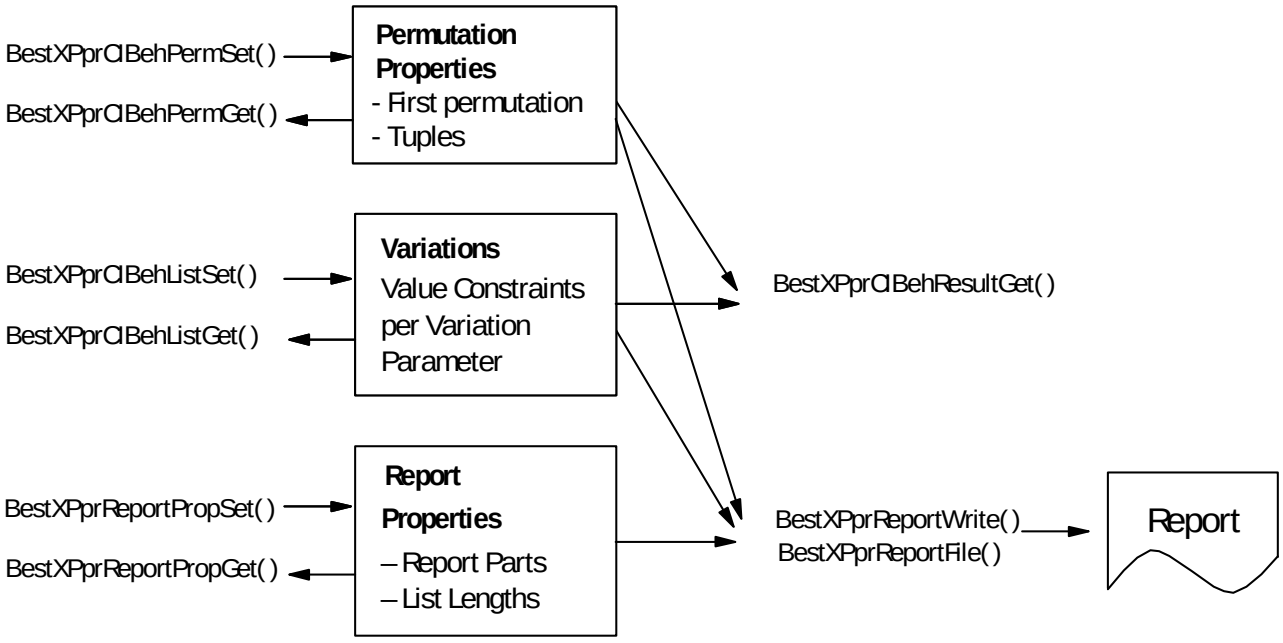
NOTE For an explanation of how the permutations are generated, refer to *"Generating Permutations" on page 123*.

Finally, the algorithm computes the number of data transfers required to achieve complete coverage by internally permutating the behavior groups against each other.

The *repetition lengths* per group and the *coverage information* per group and per group combination (tuples) can be found in the report.

How to Program CI Behavior Permutations

The following figure shows the completer-initiator behavior and report functions used to prepare and to perform the permutation of the completer-initiator behaviors.



Programming Steps

To program completer-initiator behavior permutations, the following steps are required:

1. Prepare a permutation by setting the permutation properties according to the resource requirements.

   Use *BestXPprCIBehPermSet*.

   Completer-initiator behavior permutation properties are:

   – the maximum number of groups that are permutated against each other for calculation of coverage

   – a starting point for the permutation algorithm

2. Define the lists of values for the variations with *BestXPprCIBehListSet*.

   These lists specify values for the completer-initiator behavior properties to be permutated according to the testing requirements. For the behavior properties, refer to "*bx_cibehtype*" in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference* or *Agilent E2922A/B Opt. 320 C-API/PPR Reference*.

3 If you want to check whether your test requirements are really suitable, request the test results or coverage.

Use *BestXPprCIBehResultGet*.

4 Write all PPR settings to the testcard with *BestXPprProg*.

5 The permutation results can be requested from the PCI-X Protocol Permutator and Randomizer software. Adjusted properties and permutation results can then be written to a report file.

Use *BestXPprReportWrite* or *BestXPprReportFile*.

NOTE The contents of the report file can be controlled with *BestXPprReportSet* and *BestXPprReportGet*.

Example for Programming CI Behavior Permutations

In general, the setup of the C code for programming the completer-initiator behavior permutations takes place in the same way as described for requester-initiator permutations.

Refer to *"Example for Programming RI Behavior Permutations" on page 146* for programming requester-initiator behavior permutations.

Programming RT Behavior Permutations

The requester-target behaviors can be constrained and permutated in the same way as the requester-initiator block variation parameters described above.

Requester-Target Behaviors

To achieve more sophisticated randomization opportunities, the requester-target behaviors are divided into groups, which are varied against each other. The following tables show which behaviors are assigned to which group:

Group	Behaviors
Group 0	DecSpeed, Ack64
Group 1	Initial, Latency
Group 2	Subseq, SubseqPhase

Variation Parameters

A list of values can be specified for every requester-target behavior.

Coverage

To get the coverage result, the requester-target behaviors are first permutated against required behaviors within their own group. The resulting repetition length is increased to the next higher unoccupied prime number greater than 2—the prime number 2 is skipped to obtain an odd requester-initiator behavior page size.

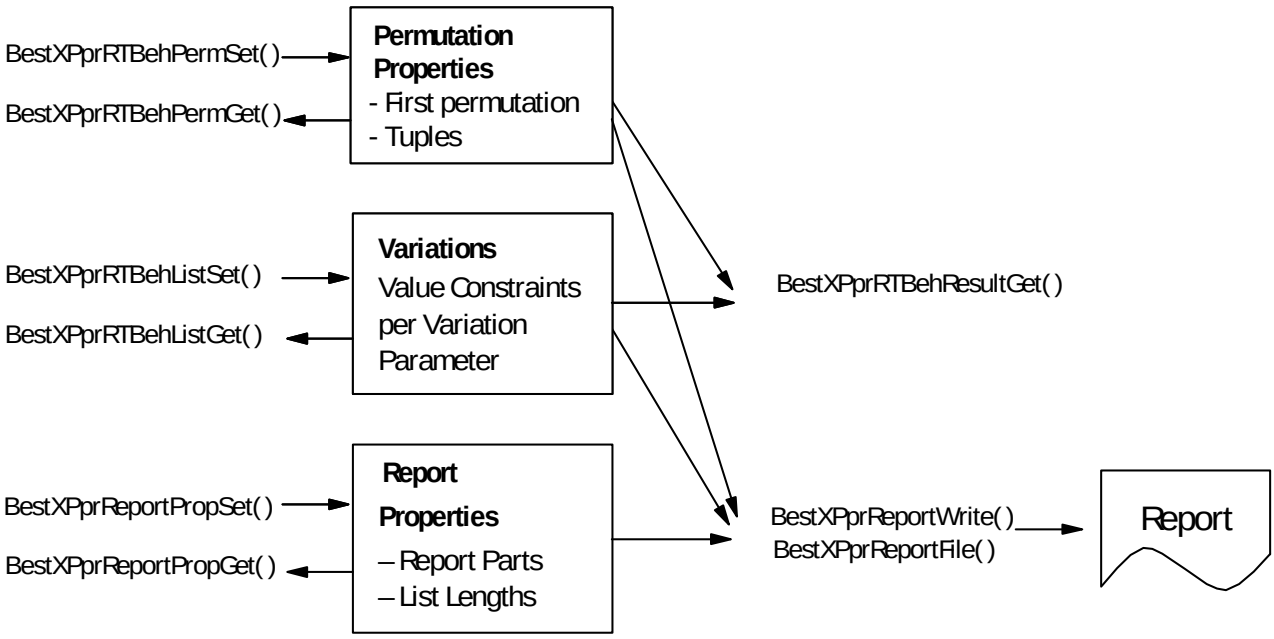
NOTE For an explanation of how the permutations are generated, refer to *"Generating Permutations" on page 123*.

Finally, the algorithm computes the number of data transfers required to achieve complete coverage by internally permutating the behavior groups against each other.

The *repetition lengths* per group and the *coverage information* per group and per group combination (tuples) can be found in the report.

How to Program RT Behavior Permutations

The following figure shows the requester-target behaviors and report functions used to prepare and to perform the permutation of the requester-target behaviors.



Programming Steps

To program requester-target behavior permutations, the following steps are required:

1. Prepare a permutation by setting the permutation properties according to the resource requirements.

   Use *BestXPprRTBehPermSet*.

   Requester-target behavior permutation properties are:

   – the maximum number of groups that are permutated against each other for calculation of coverage

   – a starting point for the permutation algorithm

2. Define the lists of values for the variations with *BestXPprRTBehListSet*.

   These lists specify values for the requester-target behavior properties to be permutated according to the testing requirements. For the behavior properties, refer to "*bx_rtbehtype*" in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference* or *Agilent E2922A/B Opt. 320 C-API/PPR Reference*.

3 If you want to check whether your test requirements are really suitable, request the test results or coverage.

Use *BestXPprRIBehResultGet*.

4 Write all PPR settings to the testcard with *BestXPprProg*.

5 The permutation results can be requested from the PCI-X Protocol Permutator and Randomizer software. Adjusted properties and permutation results can then be written to a report file.

Use *BestXPprReportWrite* or *BestXPprReportFile*.

NOTE The contents of the report file can be controlled with *BestXPprReportSet* and *BestXPprReportGet*.

Example for Programming RT Behavior Permutations

In general, the setup of the C code for programming the requester-target behavior permutations takes place in the same way as described for requester-initiator permutations.

Refer to *"Example for Programming RI Behavior Permutations" on page 146* for programming requester-initiator behavior permutations.

Generating PPR Reports

A report lists the executed protocol variations and shows which protocol attributes are permutated against which other protocol attributes after which number of data transfers.

In detail, the report contains the following information:

- Creation date and time
- Hardware, generic and report properties

- Requester-initiator block variations
- Requester-initiator behavior variations
- Requester-target behavior variations
- Completer-initiator behavior variations
- Completer-target behavior variations
- Report properties
- C-API abbreviations

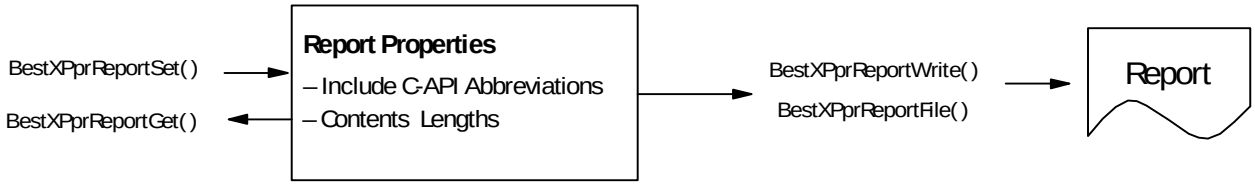
The contents of the report can be limited by setting report properties. For a list of constraints, refer to *"bppr_reportproptype"* in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference* or *Agilent E2922A/B Opt. 320 C-API/PPR Reference*.

The report can be generated and written into a specified file.

The program generated for *"Example Test Design" on page 128* can be found in *"Analyzing the Report" on page 159*.

How to Generate PPR Reports

The following figure shows the report functions used to prepare all kinds of permutations and to view the test results.



Programming Steps

Programming the report functions requires the following steps:

1. Control the contents of the report file. For the list of available properties, see *"bppr_reportproptype"* in the *Agilent E2929A/B Opt. 320 C-API/PPR Reference*.

   Use *BestXPprReportSet* and *BestXReportGet*.

2. Request the permutation results from the PCI-X Protocol Permutator and Randomizer software and write all adjusted properties and the permutation results to a report file.

   Use *BestXPprReportWrite* or *BestXPprReportFile*.

Example for Generating PPR Reports

Task Include the C-API abbreviations into the PPR report and create report files for each new permutation.

Implementation

```
/* Include the C-API abbreviations into the PPR report. */
BX_TRY(BestXPprReportSet(handle, BXPPR_REPORT_CAPI, BX_YES));
/* Create a report file name */
strcat (reportname, _itoa(i, num, 10));
strcat (reportname, ".txt");
/* Generate a report file for each new permutation (iteration) */
BX_TRY(BestXPprReportFile(handle, reportname));
```

Running a PPR Test

To run the PPR test, the PCI-X Protocol Permutation and Randomization software is not required. This is performed by the testcard's exerciser run functions. To analyze errors that occur during the test, the testcard's analyzer functions can be used.

How to Run a PPR Test

Programming Steps To run a PPR test, the following steps are required:

1. Program the PPR und Exerciser settings testcard with *BestXPprProg* and *BestXExerciserProg*.
2. Run the Exerciser with *BestXExerciserRun*.

Example for Running a PPR Test

Task Run a PPR test.

Implementation

```
/* Evaluate how many permutations are done */
BX_TRY(BestXPprRIBlkResultGet(handle, BXPPR_RIBLKRES_LASTPERM,
                              &lastperm));
BX_TRY(BestXPprRIBlkResultGet(handle, BXPPR_RIBLKRES_ACTUALSIZE,
                              &actualsize));
```

```
printf("lastperm = %ld actualsize = %d\n", lastperm, actualsize);

for (i=1; i < lastperm; i+=actualsize)

{
   char num[6];
   char reportname[80] = "c:\\temp\\PprReport";

   /* Because the block transfer memory can only hold 256 entries
   and we are creating a much larger permutation set, so we must
   iterate this to run all different combinations */

   BX_TRY(BestXPprRIBlkPermSet(handle, BXPPR_RIBLKPERM_FIRSTPERM,
                               i));

   BX_TRY(BestXPprRIBlkResultGet(handle, BXPPR_RIBLKRES_ACTUALSIZE,
                               &actualsize));

   /* Program the PPR und Exerciser settings testcard */

   BX_TRY(BestXPprProg(handle));
   BX_TRY(BestXExerciserProg(handle));

   /* Create a report file name */

   strcat (reportname, _itoa(i, num, 10));
   strcat (reportname, ".txt");

   /* Generate a report file for each new permutation (iteration) */

   BX_TRY(BestXPprReportFile(handle, reportname));
   time( &start );

   /* Start the exerciser … */

   BX_TRY(BestXExerciserRun(handle));
   time( &finish );

   /* …and run it for EXECUTION_TIME seconds per permutation list*/

   while ((difftime(finish,start)) < EXECUTION_TIME)
   time(&finish);
   BX_TRY(BestXExerciserStop(handle));

}
```

Analyzing the Report

This section contains a report created by the example C program, assuming that no errors occurred during program execution. The individual sections of the report are explained in detail. All reports produced by the PCI-X Protocol Permutator and Randomizer software are set up like this, unless some sections are suppressed by report property settings. The results in these reports could also be queried by C functions.

Report of C-API Abbreviations

The report property "Report of C-API abbreviations" (`BXPPR_REPORT_CAPI`), was active during program execution. Therefore, the C-API names of properties are given in each line. This information can be used to easily find the properties in your C program.

The reports of completer-target, completer-initiator and requester-target behavior permutations and their behavior permutations tables are not considered in the example. These reports are similar to those of the requester-initiator behavior permutations and the requester-initiator behavior table.

NOTE The order of the report sections listed here is not exactly the same as in the real PPR report. For the exact order, see *"Report Listing" on page 174*.

Report Header

The report starts with a header containing the creation date and time, followed by hardware information.

```
PCI-X Protocol Permutator & Randomizer
======================================
Report generated on 18-Mar-2002, 15:05:05 h
--------------------------------------------------------------------------------

Hardware Properties
-------------------
   HW Type ..................................................... E2929A_DEEP
   Connection .................................................. Online
```

General Properties

The general properties then follow. These are used to compute the testing times and a series of pseudo random numbers.

Refer to *"Preparing for PPR Programming" on page 131*.

```
Generic Properties
------------------
   Use RI Blk ................... BXPPR_GEN_USE_RIBLK ......... Yes
   Use RI Beh ................... BXPPR_GEN_USE_RIBEH ......... Yes
   Use RT Beh ................... BXPPR_GEN_USE_RTBEH ......... Yes
   Use CI Beh ................... BXPPR_GEN_USE_CIBEH ......... Yes
   Use CT Beh ................... BXPPR_GEN_USE_CTBEH ......... Yes
   Algorithm .................... BXPPR_GEN_ALGORITHM ......... Perm
   Preset ....................... BXPPR_GEN_PRESET ............ Default
   Level ........................ BXPPR_GEN_LEVEL ............. Data
   Bus Speed .................... BXPPR_GEN_BUSSPEED .......... 100002929
   Bus Width .................... BXPPR_GEN_BUSWIDTH .......... 64
   Seed ......................... BXPPR_GEN_SEED .............. 0
   Xfer clocks .................. BXPPR_GEN_XFERCLKS .......... 5
   ADB Limitation ............... BXPPR_GEN_ADBLIMITATION ..... Yes
```

Report Properties

The report properties then follow. These are used to specify the content of the report.

Refer to *"Generating PPR Reports" on page 155*.

```
Report Properties
-----------------
   Capi ......................... BXPPR_REPORT_CAPI ........... Yes
   Contents ..................... BXPPR_REPORT_CONTENTS ....... 50
```

Report of Block Permutations

Report of Requester-Intitiator Block Permutation

This section of the report deals with the requester-initiator block permutation. It shows which block and permutation properties are specified and which variations are constrained to which values.

```
Requester Initiator Block
-------------------------
   Direction .................... BXPPR_RIBLKPERM_DIRECTION ... read
   Bus Address Lo ............... BX_RIBLK_BUSADDR_LO ......... 536870912
   Bus Address Hi ............... BX_RIBLK_BUSADDR_HI ......... 805306368
   Internal Address ............. BX_RIBLK_INTADDR ............ 0
   Resource ..................... BX_RIBLK_RESOURCE ........... DataMem
   Block Size ................... BXPPR_RIBLKPERM_BLOCKSIZE ... 1048576
   Fill Gaps .................... BXPPR_RIBLKPERM_FILLGAPS .... Yes
   First Perm ................... BXPPR_RIBLKPERM_FIRSTPERM ... 1
   Start Offset ................. BXPPR_RIBLKPERM_STARTOFFSET . 0
   Size Limit ................... BXPPR_RIBLKPERM_SIZELIMIT ... nolimit
   Tuples ....................... BXPPR_RIBLKPERM_TUPLES ...... 3
```

Permutation Results

The following section shows the variation information and permutation results.

It shows whether coverage is achieved.

```
Variation Information
---------------------

   Bus Cmd ..................... BXPPR_RIBLK_BUSCMD .......... R = 3
       permutated, 2 values = <MemReadDWord,MemReadBlock>
       original list: <MemReadDWord,MemReadBlock,MemWrite,MemWriteBlock>
   ............................................................ covered.

   Byte Enables ................ BX_RIBLK_BYTEN .............. R = 19
       permutated, 16 values =
                  <All,1,2,Word1,4,5,6,Byte3,8,9,10,Byte2,Word0,Byte1,Byte0,None>
   ............................................................ covered.

   Alignment ................... BXPPR_RIBLK_ALIGN ........... R = 11
       permutated, 8 values = <0,1,2,3,4,5,6,7>
   ............................................................ covered.

   Number of Bytes ............. BX_RIBLK_NUMBYTES ........... R = 17
       permutated, 16 values = <1,2,3,4,5,6,7,8,16,32,64,128,256,512,1024,4096>
   ............................................................ covered.

   Relaxed Ordering ............ BX_RIBLK_RELAXORDER ......... R = 1
       fix = <Yes>
   ............................................................ covered.

   No Snoop .................... BX_RIBLK_NOSNOOP ............ R = 2
       permutated, 2 values = <No,Yes>
   ............................................................ covered.

Requester Initiator Block Result
--------------------------------
   Last Permutation ............ BXPPR_RIBLKRES_LASTPERM ..... 105
   Actual Size ................. BXPPR_RIBLKRES_ACTUALSIZE ... 256
   Number of Gaps .............. BXPPR_RIBLKRES_NUMGAPS ...... 151
   Number of Skipped Perms ..... BXPPR_RIBLKRES_NUMSKIPPED ... 0
```

Block Permutation Table

The block permutation table shows the permutations to be executed. Only a fraction of the tuples is given here. You can find a complete list of all permutation tuples in *"Report Listing" on page 174*.

```
Requester Initiator Block Variation
-----------------------------------
  Number of Permutations ..................................... 21318
-----------------------------------
     |                   N
   P |                   u    N
   e |                   m    o
   r |         B    A    B    S
   m |         y    l    y    n
   n |    C    t    i    t    o
   u |    m    e    g    e    o
   m |    d    n    n    s    p
-----------------------------------
   1 |    6    0    0    1    0
   2 |   14    1    1    2    1
   3 |    6    2    2    3    0
   4 |    6    3    3    4    1
   5 |   14    4    4    5    0
   6 |    6    5    5    6    1
   7 |    6    6    6    7    0
   8 |   14    7    7    8    1
   9 |    6    8    0   16    0
  10 |    6    9    1   32    1
  11 |   14   10    2   64    0
  12 |    6   11    0  128    1
  13 |    6   12    1  256    0
  14 |   14   13    2  512    1
  15 |    6   14    3 1024    0
  16 |    6   15    4 4096    1
  17 |   14    0    5    1    0
  18 |    6    1    6    1    1
  19 |    6    2    7    2    0

  …

  36 |    6    0    2    2    1
  37 |    6    1    3    3    0
  38 |   14    2    4    4    1
  39 |    6    0    5    5    0
  40 |    6    1    6    6    1
  41 |   14    2    7    7    0
  42 |    6    3    0    8    1
  43 |    6    4    1   16    0
  44 |   14    5    2   32    1
  45 |    6    6    0   64    0
  46 |    6    7    1  128    1
  47 |   14    8    2  256    0
  48 |    6    9    3  512    1
  49 |    6   10    4 1024    0
  50 |   14   11    5 4096    1

   Printout ended due to user setting of contents
```

The columns contain the permutation number, the bus commands (6 or 14) used, the byte enable, the alignment, the number of bytes transferred with the permutation and the nosnoop bit.

NOTE The PCI-X Protocol Permutation and Randomization software varies all parameters simultaneously. If it varied only one parameter from one permutation to the next while fixing the remaining, it would be possible that parameters are not varied at all.

Another reason why the PCI-X PPR software changes all parameters simultaneously is to achieve a good mix of test cases.

In the requester-initiator block variation list shown above, all block properties are permutated independently according to their value lists.

All variation list lengths of the properties are prime numbers. If they were not, the software would raise them up to the next unoccupied prime number (for an explanation of how repetition lengths are calculated, refer to *"Generating Permutations" on page 123*).

The repetition length R is the length between two permutations with equivalent tuples. A block property has taken all values after R data transfers. For example: Three data transfers are necessary to test three blocksizes.

The number of data transfers required to cover all combinations of block property values is calculated by multiplying the numbers of values of each property. For example: To combine all address alignments with all blocksizes, you need 11 × 17= 187 data transfers. To combine all address alignments with all commands and blocksizes, you need 11 × 3 × 17 = 561 transfers.

The following table shows how many block permutations are required to permutate the block properties (the tuples and the whole testing area), as required by the example test design:

Tuple	Repetition Lengths
(ALIGNMENT)	11
(BLOCKSIZE)	17
(COMMAND)	3
(BYTEN)	19
(NOSNOOP)	2
(ALIGNMENT, BLOCKSIZE, COMMAND, BYTEN, NOSNOOP)	11 × 17 × 3 × 19 × 2= 21318
R(BLOCK)	Max = 21318

Block Fitting List

The blocks contained in the block permutation table must be arranged to fit into the compound block. The compound blocksize (CBS) is determined by the resources. For the example test, it is assumed to be 1MB.

Therefore, the algorithm sequentially steps through the block permutation table and fits the individual permutations into the compound block, regarding their alignment and size.

It proceeds by filling up the block, alternating from the start and from the end of the block until all permutations are inserted. If some permutations do not fit in, it terminates.

The goal is to fit in as many permutations as possible. The FILLGAPS property can be set to enable the filling of gaps remaining between blocks with additional block transfers. This ensures that the compound block will be transferred completely.

For the example test, the algorithm can fit all blocks into the compound block of 1MB.

The report then produces the Block Fitting List, which shows the compound block resulting from the rearrangement of the example test.

```
RI Block Fitting List
-------------------------------------------------------------------------
PermNum | Start Addr  End Addr     Size Alignment Byten Command
-------------------------------------------------------------------------
   1 | 20000000\h  20000000\h       1     0   0000\b MemReadDWord
   2 | 20000001\h  20000002\h       2     1   0001\b  MemReadBlock
fill | 20000001\h  20000002\h       2  (  1)  1110\b  MemReadDWord
   4 | 20000003\h  20000006\h       4     3   0011\b  MemReadDWord
fill | 20000003\h  20000006\h       4  (  3)  1100\b  MemReadDWord
   8 | 20000007\h  2000000e\h       8     7   0111\b  MemReadBlock
fill | 20000007\h  2000000e\h       8  (  7)  1000\b  MemReadDWord
fill | 2000000f\h  2000007f\h     113  ( 15)  0000\b  MemReadDWord
   9 | 20000080\h  2000008f\h      16     0   1000\b  MemReadDWord
fill | 20000080\h  2000008f\h      16  (  0)  0111\b  MemReadDWord
fill | 20000090\h  20000100\h     113  ( 16)  0000\b  MemReadDWord
  13 | 20000101\h  20000200\h     256     1   1100\b  MemReadDWord
fill | 20000101\h  20000200\h     256  (  1)  0011\b  MemReadDWord
  35 | 20000201\h  20000201\h       1     1   1111\b  MemReadBlock
fill | 20000201\h  20000201\h       1  (  1)  0000\b  MemReadDWord
  14 | 20000202\h  20000401\h     512     2   1101\b  MemReadBlock
fill | 20000202\h  20000401\h     512  (  2)  0010\b  MemReadDWord
  69 | 20000402\h  20000402\h       1     2   1011\b  MemReadDWord
fill | 20000402\h  20000402\h       1  (  2)  0100\b  MemReadDWord
  15 | 20000403\h  20000802\h    1024     3   1110\b  MemReadDWord
fill | 20000403\h  20000802\h    1024  (  3)  0001\b  MemReadDWord
 103 | 20000803\h  20000803\h       1     3   0111\b  MemReadDWord
fill | 20000803\h  20000803\h       1  (  3)  1000\b  MemReadDWord
  16 | 20000804\h  20001803\h    4096     4   1111\b  MemReadDWord
  27 | 20000804\h  20000823\h      32     4   0111\b  MemReadDWord
fill | 20000804\h  20000823\h      32  (  4)  1000\b  MemReadDWord
fill | 20000824\h  20000880\h      93  ( 36)  0000\b  MemReadDWord

   …

fill | 200fff82\h  200fff84\h       3  (  2)  1101\b  MemReadDWord
   6 | 200fff85\h  200fff8a\h       6     5   0101\b  MemReadDWord
fill | 200fff85\h  200fff8a\h       6  (  5)  1010\b  MemReadDWord
fill | 200fff8b\h  200fffff\h     117  ( 11)  0000\b  MemReadDWord
```

Summarizing the Results

The results for coverage and testing time of the requester-initiator block permutations are described as follows:

Coverage The following table shows the repetition length for each tuple and whether it is covered after the Requester-Initiator Block Last Permutation. Because the number of the last permutation is 105, the testing goal for the BLOCK testing area is not achieved.

Tuple	Repetition Length R	Coverage
(ALIGNMENT)	11	yes
(BLOCKSIZE)	17	yes
(COMMAND)	3	yes
(BYTEN)	19	yes
(NOSNOOP)	2	yes
(ALIGNMENT, BLOCKSIZE, COMMAND, BYTEN, NOSNOOP)	11 × 17 × 3 × 19 × 2= 21318	no
R(BLOCK)	Max = 21318	no

Testing Time In the example test, the block test is performed after the 250000 dwords (=1MB compound block size) have been transferred, that is after the 21318 block transfers in the example. Assuming an average of 10 clock cycles for each of the 250000 data transfers, circa 2.5 s are needed for the data transfer (clock is 100 MHz, that is 1.0 ns per clock cycle).

Testing Time	Calculation	Value
Total T(BLOCK)	Data transfer time: 250000 × 10 clocks × 1.0 ns	2.5 s

Report of Requester-Initiator Behavior Permutation

This section reports the specified requester-initiator permutation parameters.

```
Requester Initiator Behavior Perm
---------------------------------
   First Perm ................... BXPPR_BEHPERM_FIRSTPERM ..... 1
   Tuples ....................... BXPPR_BEHPERM_TUPLES ........ 3
```

Variation Information and Permutation Results

The following subsection shows the variation constraints for the requester-initiator permutation and whether coverage is achieved.

It reports which behaviors of each group are permutated. Note that the repetition length of each group is raised up to the next prime.

```
Variation Information
---------------------

   Group 0 ...................... HW Groups G6 ................ R = 7
    Queue        : permutated, 2 values = <qa,qb>
    Steps        : fix = <0>
    Req64        : permutated, 2 values = <No,Yes>
   ............................................................ covered.

   Group 1 ...................... HW Groups G8, G9 ............ R = 5
    ByteCount    : permutated, 5 values = <33,64,72,128,4096>
   ............................................................ covered.

   Group 2 ...................... HW Groups G3 ................ R = 11
    Disconnect   : permutated, 7 values = <1,2,3,4,5,6,7>
   ............................................................ covered.

   Group 3 ...................... HW Groups G1, G2, G4, G5 .... R = 3
    Delay        : permutated, 3 values = <100,200,300>
    RelReq       : fix = <2047>
   ............................................................ covered.


Requester Initiator Behavior Result
-----------------------------------
   Last Permutation ............. BXPPR_BEHRES_LASTPERM ....... 1155
   Last tuples Permutation ...... BXPPR_BEHRES_TUPLES_LASTPERM  385
   Data ......................... BXPPR_BEHRES_DATA ........... 1014783
   Tuples Data .................. BXPPR_BEHRES_TUPLES_DATA .... 338261
   Runs ......................... BXPPR_BEHRES_RUNS ........... 1
   Tuples Runs .................. BXPPR_BEHRES_TUPLES_RUNS .... 1
```

As for the block properties permutation, the algorithm used by the PPR software ensures that all behaviors will have taken all their values at least once after `R` data phases, where `R` is the corresponding repetition length of the behavior tuples.

In the example test design, the following variation parameters have been specified for the requester-initiator behavior testing area: queue, steps, req64, byte count, disconnect, delay and relreq (see *"Permutations to be Covered" on page 129*).

Computing Repetition Lengths

The repetition lengths are computed by the algorithm as described in *"Generating Permutations" on page 123*.

To guarantee that all repetition lengths are distinct and have no common factor, each individual length is raised up to the next distinct prime number. This is necessary to prevent the algorithm from cycling through permutations with equivalent tuples.

NOTE 2 is not used in this algorithm as a repetition length, even though it is a prime number. This guarantees that only behavior pages of odd lengths are generated (which is necessary for permutating behaviors against blocks, see below).

The software first groups the behaviors and performs a complete permutation within the group each behavior belongs to. Afterwards each behavior is permutated through all of its possible values, similar to the permutation of block properties.

Requester-Initiator Behavior Permutation Table

After running the permutation algorithm, the behaviors are permutated as shown in the following report section. Note that only permutated (not fixed) behavior parameters are listed.

```
Requester Initiator Behavior Variation
--------------------------------------
  Number of Permutations ..................................... 1155

-----------------------------------
     |              D
     |         B    i
     |         y    s
   P |         t    c
   e |         e    o
   r |    Q    C    n    D    R
   m |    u    o    n    e    e
   n |    e    u    e    l    q
   u |    u    n    c    a    6
   m |    e    t    t    y    4
-----------------------------------
   1 |    1   33    1  100    0
   2 |    2   64    2  200    1
   3 |    1   72    3  300    0
   4 |    1  128    4  100    1
   5 |    2 4096    5  200    0
   6 |    1   33    6  300    1
   7 |    1   64    7  100    0
   8 |    2   72    1  200    1
   9 |    1  128    2  300    0
  10 |    1 4096    3  100    1
  11 |    2   33    4  200    0
  12 |    1   64    5  300    1
  13 |    1   72    6  100    0
  14 |    2  128    7  200    1
  15 |    1 4096    1  300    0
  16 |    1   33    2  100    1
  17 |    2   64    3  200    0
  18 |    1   72    4  300    1
  19 |    1  128    5  100    0
  20 |    2 4096    6  200    1
  21 |    1   33    7  300    0
  22 |    1   64    1  100    1
  23 |    2   72    2  200    0
  24 |    1  128    3  300    1
  25 |    1 4096    4  100    0

   …

  43 |    1   72    1  100    0
  44 |    2  128    2  200    1
  45 |    1 4096    3  300    0
  46 |    1   33    4  100    1
  47 |    2   64    5  200    0
  48 |    1   72    6  300    1
  49 |    1  128    7  100    0
  50 |    2 4096    1  200    1

   Printout ended due to user setting of contents
```

The list shows the algorithm's operating principles. (It is limited to the first 50 permutations (of 1155 overall)).

NOTE Permutations 26 to 42 are skipped in this printout.

Only the varying parameters are included. The length of the table printout may be restricted by the report properties (50 lines are reported).

Summarizing the Results

The results for coverage and testing time of the requester-initiator block permutations are described as follows:

Coverage In the example test, complete coverage is achieved.

The following table shows the repetition length for each tuple and whether it is covered after the Requester-Initiator Behavior Last Permutation. Thus the testing goal for the Behavior testing area is achieved.

Tuple	Behaviors	Repetition Length R	Coverage
Group 0	Queue, Steps, Req64	7	yes
Group 1	ByteCount	5	yes
Group 2	Diconnect	11	yes
Group 3	Delay, RelReq	3	yes
R(All Groups)		Max = 7 x 5 x 11 x 3 = 1155	yes

Testing Time IThe testing time can be calculated by the testing time for the requester-initiator block permutations and the number of requester-initiator behavior permutations.

Testing Time	Calculation	Value
Total T	Testing Time for RI Block Permutations (= 2.5 s) x Number of Behavior Permutations (=1155)	2887.5 s

Report of Requester-Initiator Block vs. Requester-Initiator Behavior Permutation

The following lines in the report result from the permutation of Alignment (block property) vs. Bytecount (behavior property).

```
Bytecount/Alignment Information
-------------------------------
Length of bytecount list 5
Bytecount List: 33, 64, 72, 128, 4096
  33 (32 bit):     9     9     9     9
     (64 bit):     5     5     5     5     5     5     5     5
  64 (32 bit):    16    17    17    17
     (64 bit):     8     9     9     9     9     9     9     9
  72 (32 bit):    18    19    19    19
     (64 bit):     9    10    10    10    10    10    10    10
 128 (32 bit):    32    33    33    33
     (64 bit):    16    17    17    17    17    17    17    17
4096 (32 bit):  1024  1025  1025  1025
     (64 bit):   512   513   513   513   513   513   513   513
All possible number of data phases:
    5    8    9   10   16   17   18   19   32   33  512  513 1024 1025
Total number of different transfer lengths: 14

Period Byte Count: 89480\h
Period Iterations: 639

Minimum Byte Count to reach all alignments: 43938\h
Minimum Iterations to reach all alignments: 315

Minimum Byte count 43938\h reached after 316 iterations.

After 316 iterations possible alignments are

0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30
,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,
58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,8
5,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,108,10
9,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,
After 316 iterations alignments not covered are
```

Further Options and Possibilities

The Protocol Permutator and Randomizer software provides further options and possibilities that were not covered by the example scenario and therefore have not yet been explained.

Optimizing Testing Time

The testing time can be optimized by taking the following into account:

- Keep the byte counts and the number of bursts small.

  Very long bursts result in long data transfer times, even if they are varied against each other or other parameters.

- Include short bursts in the variation list only if exactly these short bursts are to be examined.

  In most cases, short bursts are added to the block permutation memory automatically in order to fill gaps, or to increase the number of bursts to a prime number.

- Vary either byte counts or block size.

- Vary only behaviors of interest.

General Tips

Regard the following general tips:

- Avoid adding exceptions, such as asserting system errors to permutations, if the system under test is unable to handle them.

- If the target terminates with a disconnect, it is not guaranteed that a burst with a desired length has been covered.

  For the coverage computations, it is assumed that it is sufficient to know that the target would have been ready for a burst of that length.

- The test cannot guarantee the coverage of errors due to combination of PCI-X protocol errors and internal states of the device under test.

  However, the test can be used to stress the device under test with the same permutation sequences multiple times, while the device under test independently passes different internal states.

Presetting Values

To avoid unexpected program behavior, default values can be set. These values are preset after initialization of the PCI-X Protocol Permutation and Randomization software or parts of it by means of the `...Init` functions.

After initialization, the default values can be set with the `...DefaultSet` functions.

Byte Enable Variation

Byte enables can be varied like the other parameters, but note that if `FILLGAPS` is activated, block transfers may be added to ensure that all byte enables were used after the compound block was transferred. In this case, variations of other parameters may be used with value variations which, perhaps intentionally, were not specified.

Uncovered Permutations

If the coverage of permutated tuples is not achieved (although it is required), the report will contain a hint. In most cases, increasing the system resources will help.

If resources cannot be increased, you can try to take advantage of the following properties provided by the PCI-X Protocol Permutation and Randomization software:

- `...LASTPERM`, which contains the number of the last permutation that could be covered.
- `...FIRSTPERM`, which allows the setting of the number of the permutation where the algorithm should start.

These properties can be used to fill the different pages with behaviors or blocks. The algorithm can be set to continue where the previous invocation stopped.

Report Listing

```
PCI-X Protocol Permutator & Randomizer

====================================

Report generated on 18-Mar-2002, 15:05:05 h

-----------------------------------------------------------------

Hardware Properties

-------------------

    HW Type ........................................... E2929A_DEEP
    Connection ........................................ Online


Generic Properties
------------------

Use RI Blk ................... BXPPR_GEN_USE_RIBLK ...... Yes
Use RI Beh ................... BXPPR_GEN_USE_RIBEH ...... Yes
Use RT Beh ................... BXPPR_GEN_USE_RTBEH ...... Yes
Use CI Beh ................... BXPPR_GEN_USE_CIBEH ...... Yes
Use CT Beh ................... BXPPR_GEN_USE_CTBEH ...... Yes
Algorithm .................... BXPPR_GEN_ALGORITHM ...... Perm
Preset ....................... BXPPR_GEN_PRESET ......... Default
Level ........................ BXPPR_GEN_LEVEL .......... Data
Bus Speed .................... BXPPR_GEN_BUSSPEED ....... 100002929
Bus Width .................... BXPPR_GEN_BUSWIDTH ....... 64
Seed ......................... BXPPR_GEN_SEED ........... 0
Xfer clocks .................. BXPPR_GEN_XFERCLKS ....... 5
ADB Limitation ............... BXPPR_GEN_ADBLIMITATION .. Yes


Report Properties
-----------------

Capi ......................... BXPPR_REPORT_CAPI ........... Yes
Contents ..................... BXPPR_REPORT_CONTENTS ....... 50
-------------------------------------------------------------------------------

Requester Initiator Block Variation
===================================

Requester Initiator Block
-------------------------
Direction .................... BXPPR_RIBLKPERM_DIRECTION ... read
Bus Address Lo ............... BX_RIBLK_BUSADDR_LO ...... 536870912
Bus Address Hi ............... BX_RIBLK_BUSADDR_HI ......... 805306368
Internal Address ............. BX_RIBLK_INTADDR ............ 0
Resource ..................... BX_RIBLK_RESOURCE ........ DataMem
Block Size ................... BXPPR_RIBLKPERM_BLOCKSIZE ... 1048576
Fill Gaps .................... BXPPR_RIBLKPERM_FILLGAPS .... Yes
First Perm ................... BXPPR_RIBLKPERM_FIRSTPERM ... 1
Start Offset ................. BXPPR_RIBLKPERM_STARTOFFSET . 0
Tuples ....................... BXPPR_RIBLKPERM_TUPLES ...... 3
```

```
Variation Information
---------------------

Bus Cmd ..................... BXPPR_RIBLK_BUSCMD .......... R = 3

       permutated, 2 values = <MemReadDWord,MemReadBlock>

       original list: <MemReadDWord,MemReadBlock,MemWrite,MemWriteBlock>
.......................................................... covered.

Byte Enables ................ BX_RIBLK_BYTEN .............. R = 19
       permutated, 16 values =
<All,1,2,Word1,4,5,6,Byte3,8,9,10,Byte2,Word0,Byte1,Byte0,None>
.......................................................... covered.
Alignment ................... BXPPR_RIBLK_ALIGN ........... R = 11
       permutated, 8 values = <0,1,2,3,4,5,6,7>

   ..................................................... covered.

Number of Bytes ............. BX_RIBLK_NUMBYTES ........... R = 17

       permutated, 16 values = <1,2,3,4,5,6,7,8,16,32,64,128,256,512,1024,4096>
.......................................................... covered.

Relaxed Ordering ............. BX_RIBLK_RELAXORDER ......... R = 1
       fix = <Yes>.............................................................
covered.

No Snoop ..................... BX_RIBLK_NOSNOOP ............ R = 2
       permutated, 2 values =
<No,Yes>..................................................... covered.


RI Block Fitting List

-------------------------------------------------------------------
PermNum | Start Addr  End Addr     Size Alignment Byten Command
-------------------------------------------------------------------

    1 | 20000000\h  20000000\h       1     0   0000\b MemReadDWord
    2 | 20000001\h  20000002\h       2     1   0001\b  MemReadBlock
fill | 20000001\h  20000002\h       2  (  1)  1110\b  MemReadDWord
    4 | 20000003\h  20000006\h       4     3   0011\b  MemReadDWord
fill | 20000003\h  20000006\h       4  (  3)  1100\b  MemReadDWord
    8 | 20000007\h  2000000e\h       8     7   0111\b  MemReadBlock
fill | 20000007\h  2000000e\h       8  (  7)  1000\b  MemReadDWord
fill | 2000000f\h  2000007f\h     113  ( 15)  0000\b  MemReadDWord
    9 | 20000080\h  2000008f\h      16     0   1000\b  MemReadDWord
fill | 20000080\h  2000008f\h      16  (  0)  0111\b  MemReadDWord
fill | 20000090\h  20000100\h     113  ( 16)  0000\b  MemReadDWord
  13 | 20000101\h  20000200\h     256     1   1100\b  MemReadDWord
fill | 20000101\h  20000200\h     256  (  1)  0011\b  MemReadDWord
  35 | 20000201\h  20000201\h       1     1   1111\b  MemReadBlock
fill | 20000201\h  20000201\h       1  (  1)  0000\b  MemReadDWord
  14 | 20000202\h  20000401\h     512     2   1101\b  MemReadBlock
fill | 20000202\h  20000401\h     512  (  2)  0010\b  MemReadDWord
  69 | 20000402\h  20000402\h       1     2   1011\b  MemReadDWord
fill | 20000402\h  20000402\h       1  (  2)  0100\b  MemReadDWord
  15 | 20000403\h  20000802\h    1024     3   1110\b  MemReadDWord
fill | 20000403\h  20000802\h    1024  (  3)  0001\b  MemReadDWord
 103 | 20000803\h  20000803\h       1     3   0111\b  MemReadDWord
fill | 20000803\h  20000803\h       1  (  3)  1000\b  MemReadDWord
  16 | 20000804\h  20001803\h    4096     4   1111\b  MemReadDWord
  27 | 20000804\h  20000823\h      32     4   0111\b  MemReadDWord
fill | 20000804\h  20000823\h      32  (  4)  1000\b  MemReadDWord
fill | 20000824\h  20000880\h      93  ( 36)  0000\b  MemReadDWord
…
```

```
fill | 200fff82\h  200fff84\h       3  (  2)  1101\b  MemReadDWord
   6 | 200fff85\h  200fff8a\h       6     5   0101\b  MemReadDWord
fill | 200fff85\h  200fff8a\h       6  (  5)  1010\b  MemReadDWord
fill | 200fff8b\h  200fffff\h     117  ( 11)  0000\b  MemReadDWord


Requester Initiator Block Result
--------------------------------
Last Permutation ............. BXPPR_RIBLKRES_LASTPERM ..... 105
Actual Size .................. BXPPR_RIBLKRES_ACTUALSIZE ... 256
Number of Gaps ............... BXPPR_RIBLKRES_NUMGAPS ...... 151
Number of Skipped Perms ...... BXPPR_RIBLKRES_NUMSKIPPED ... 0


Requester Initiator Block Variation
-----------------------------------
Number of Permutations ..................................... 21318
-----------------------------------
      |                 N
    P |                 u    N
    e |                 m    o
    r |        B    A    B    S
    m |        y    l    y    n
    n |    C   t    i    t    o
    u |    m   e    g    e    o
    m |    d   n    n    s    p
-----------------------------------
    1 |    6   0    0    1    0
    2 |   14   1    1    2    1
    3 |    6   2    2    3    0
    4 |    6   3    3    4    1
    5 |   14   4    4    5    0
    6 |    6   5    5    6    1
    7 |    6   6    6    7    0
    8 |   14   7    7    8    1
    9 |    6   8    0   16    0
   10 |    6   9    1   32    1
   11 |   14  10    2   64    0
   12 |    6  11    0  128    1
   13 |    6  12    1  256    0
   14 |   14  13    2  512    1
   15 |    6  14    3 1024    0
   16 |    6  15    4 4096    1
   17 |   14   0    5    1    0
   18 |    6   1    6    1    1
   19 |    6   2    7    2    0
   20 |   14   0    0    3    1
   21 |    6   1    1    4    0
   22 |    6   2    2    5    1
   23 |   14   3    0    6    0
```

```
  24 |    6    4    1    7    1
  25 |    6    5    2    8    0
  26 |   14    6    3   16    1
  27 |    6    7    4   32    0
  28 |    6    8    5   64    1
  29 |   14    9    6  128    0
  30 |    6   10    7  256    1
  31 |    6   11    0  512    0
  32 |   14   12    1 1024    1
  33 |    6   13    2 4096    0
  34 |    6   14    0    1    1
  35 |   14   15    1    1    0
  36 |    6    0    2    2    1
  37 |    6    1    3    3    0
  38 |   14    2    4    4    1
  39 |    6    0    5    5    0
  40 |    6    1    6    6    1
  41 |   14    2    7    7    0
  42 |    6    3    0    8    1
  43 |    6    4    1   16    0
  44 |   14    5    2   32    1
  45 |    6    6    0   64    0
  46 |    6    7    1  128    1
  47 |   14    8    2  256    0
  48 |    6    9    3  512    1
  49 |    6   10    4 1024    0
  50 |   14   11    5 4096    1

    Printout ended due to user setting of contents
--------------------------------------------------------------------------------

Requester Initiator Behavior Variation
======================================

Requester Initiator Behavior Perm
---------------------------------
   First Perm ................... BXPPR_BEHPERM_FIRSTPERM ..... 1
   Tuples ....................... BXPPR_BEHPERM_TUPLES ........ 3
```

```
Variation Information
---------------------

   Group 0 ..................... HW Groups G6 ............. R = 7
    Queue        : permutated, 2 values = <qa,qb>
    Steps        : fix = <0>
    Req64        : permutated, 2 values = <No,Yes>
   ........................................................... covered.

   Group 1 ..................... HW Groups G8, G9 ......... R = 5
    ByteCount    : permutated, 5 values = <33,64,72,128,4096>
   ........................................................... covered.

   Group 2 ..................... HW Groups G3 ............. R = 11
    Disconnect   : permutated, 7 values = <1,2,3,4,5,6,7>
   ........................................................... covered.

   Group 3 ..................... HW Groups G1, G2, G4, G5 .. R = 3
    Delay        : permutated, 3 values = <100,200,300>
    RelReq       : fix = <2047>
   ........................................................ covered.

Requester Initiator Behavior Result
-----------------------------------
   Last Permutation ............. BXPPR_BEHRES_LASTPERM ...... 1155
   Last tuples Permutation ...... BXPPR_BEHRES_TUPLES_LASTPERM  385
   Data ......................... BXPPR_BEHRES_DATA ....... 1014783
   Tuples Data .................. BXPPR_BEHRES_TUPLES_DATA . 338261
   Runs ......................... BXPPR_BEHRES_RUNS ........... 1
   Tuples Runs .................. BXPPR_BEHRES_TUPLES_RUNS .... 1
```

```
Requester Initiator Behavior Variation
--------------------------------------
   Number of Permutations ................................. 1155

----------------------------------
      |             D
      |        B    i
      |        y    s
    P |        t    c
    e |        e    o
    r |    Q    C    n    D    R
    m |    u    o    n    e    e
    n |    e    u    e    l    q
    u |    u    n    c    a    6
    m |    e    t    t    y    4
----------------------------------
    1 |    1   33    1  100    0
    2 |    2   64    2  200    1
    3 |    1   72    3  300    0
    4 |    1  128    4  100    1
    5 |    2 4096    5  200    0
    6 |    1   33    6  300    1
    7 |    1   64    7  100    0
    8 |    2   72    1  200    1
    9 |    1  128    2  300    0
   10 |    1 4096    3  100    1
   11 |    2   33    4  200    0
   12 |    1   64    5  300    1
   13 |    1   72    6  100    0
   14 |    2  128    7  200    1
   15 |    1 4096    1  300    0
   16 |    1   33    2  100    1
   17 |    2   64    3  200    0
   18 |    1   72    4  300    1
   19 |    1  128    5  100    0
   20 |    2 4096    6  200    1
   21 |    1   33    7  300    0
   22 |    1   64    1  100    1
   23 |    2   72    2  200    0
   24 |    1  128    3  300    1
   25 |    1 4096    4  100    0
   26 |    2   33    5  200    1
   27 |    1   64    6  300    0
   28 |    1   72    7  100    1
```

```
29 |    2  128    1  200    0
30 |    1 4096    2  300    1
31 |    1   33    3  100    0
32 |    2   64    4  200    1
33 |    1   72    5  300    0
34 |    1  128    6  100    1
35 |    2 4096    7  200    0
36 |    1   33    1  300    1
37 |    1   64    2  100    0
38 |    2   72    3  200    1
39 |    1  128    4  300    0
40 |    1 4096    5  100    1
41 |    2   33    6  200    0
42 |    1   64    7  300    1
43 |    1   72    1  100    0
44 |    2  128    2  200    1
45 |    1 4096    3  300    0
46 |    1   33    4  100    1
47 |    2   64    5  200    0
48 |    1   72    6  300    1
49 |    1  128    7  100    0
50 |    2 4096    1  200    1

Printout ended due to user setting of contents
```

```
Bytecount/Alignment Information
------------------------------
Length of bytecount list 5
Bytecount List: 33, 64, 72, 128, 4096
  33 (32 bit):     9     9     9     9
     (64 bit):     5     5     5     5     5     5     5     5
  64 (32 bit):    16    17    17    17
     (64 bit):     8     9     9     9     9     9     9     9
  72 (32 bit):    18    19    19    19
     (64 bit):     9    10    10    10    10    10    10    10
 128 (32 bit):    32    33    33    33
     (64 bit):    16    17    17    17    17    17    17    17
4096 (32 bit):  1024  1025  1025  1025
     (64 bit):   512   513   513   513   513   513   513   513
All possible number of data phases:
    5    8    9   10   16   17   18   19   32   33  512  513 1024 1025
Total number of different transfer lengths: 14

Period Byte Count: 89480\h
Period Iterations: 639

Minimum Byte Count to reach all alignments: 43938\h
Minimum Iterations to reach all alignments: 315

Minimum Byte count 43938\h reached after 316 iterations.

After 316 iterations possible alignments are

0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,
31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58
,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,8
6,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,108,109,11
0,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,
After 316 iterations alignments not covered are
-------------------------------------------------------------------
```

```
Requester Target Behavior Variation
===================================
Requester Target Behavior
-------------------------
   First Perm ................... BXPPR_BEHPERM_FIRSTPERM ..... 1
   Tuples ....................... BXPPR_BEHPERM_TUPLES ........ 3

Variation Information
---------------------

   Group 0 ...................... HW Groups G4 .............. R = 1
    DecSpeed     : fix = <B>
    Ack64        : fix = <Yes>
   ..................................................... covered.

   Group 1 ...................... HW Groups G1 .............. R = 1
    Initial      : fix = <Accept>
    Latency      : fix = <3>
   ..................................................... covered.

   Group 2 ...................... HW Groups G2, G3 .......... R = 1
    Subseq       : fix = <Accept>
    SubseqPhase  : fix = <0>
   ..................................................... covered.

Requester Target Behavior Result
--------------------------------
   Last Permutation ............. BXPPR_BEHRES_LASTPERM ....... 1
   Last tuples Permutation ...... BXPPR_BEHRES_TUPLES_LASTPERM  1
Requester Target Behavior Variation
-----------------------------------
   Number of Permutations ..................................... 1
<not permutated>

-----------------------------------------------------------------
```

```
Completer Initiator Behavior Variation
======================================
Completer Initiator Behavior
----------------------------
   First Perm ................... BXPPR_BEHPERM_FIRSTPERM ..... 1
   Tuples ....................... BXPPR_BEHPERM_TUPLES ........ 3
Variation Information
---------------------
Group 0 ..................... HW Groups G8 ............... R = 1
    Queue        : fix = <Next>
   ..................................................... covered.
Group 1 ..................... HW Groups G7 ............... R = 1
    ErrMsg       : fix = <No>
   ..................................................... covered.
Group 2 ..................... HW Groups G3 ............... R = 1
    Partition    : fix = <No>
   ..................................................... covered.
Group 3 ..................... HW Groups G1, G2, G4, G5 .. R = 1
    Delay        : fix = <1>
    RelReq       : fix = <2047>
   ..................................................... covered.
Group 4 ..................... HW Groups G6 ............... R = 1
    Steps        : fix = <2>
    Req64        : fix = <Yes>
   ..................................................... covered.

Completer Initiator Behavior Result
-----------------------------------
   Last Permutation ............. BXPPR_BEHRES_LASTPERM ....... 1
   Last tuples Permutation ...... BXPPR_BEHRES_TUPLES_LASTPERM  1

Completer Initiator Behavior Variation
--------------------------------------
   Number of Permutations ................................... 1
<not permutated>
-----------------------------------------------------------------------

Completer Target Behavior Variation
===================================
Completer Target Behavior
-------------------------
   First Perm ................... BXPPR_BEHPERM_FIRSTPERM ..... 1
   Tuples ....................... BXPPR_BEHPERM_TUPLES ........ 3
```

```
Variation Information
---------------------
Group 0 ..................... HW Groups G4 ............. R = 1
    DecSpeed     : fix = <B>
    Ack64        : fix = <Yes>
    SplitLatency : fix = <3>
   ..................................................... covered.
Group 1 ..................... HW Groups G1 ............. R = 1
    Initial      : fix = <Accept>
    Latency      : fix = <3>
   ..................................................... covered.
Group 2 ..................... HW Groups G2, G3 ......... R = 1
    Subseq       : fix = <Accept>
    SubseqPhase  : fix = <0>
    SplitEnable  : fix = <Yes>
   ..................................................... covered.

Completer Target Behavior Result
--------------------------------
   Last Permutation ............. BXPPR_BEHRES_LASTPERM ....... 1
   Last tuples Permutation ...... BXPPR_BEHRES_TUPLES_LASTPERM  1

Completer Target Behavior Variation
-----------------------------------
   Number of Permutations ................................... 1
<not permutated>

-----------------------------------------------------------------
<End of PPR Report>
```

Code Listing

WARNING

This program fragment writes data to the system memory. To run this program in a real environment, a line that allocates the required memory must be added.

```
#include <stdafx.h>
#include <xpciapi.h>
#include <pprx.h>
#include "SetupUtil.h"
#include <time.h>

#define EXECUTION_TIME 5

int main(int argc, char* argv[])

{

   BX_TRY_VARS_NO_PROG;
   // additional local variable declarations, here

   bx_handletype handle;
   time_t start, finish;
   bx_int32 width, speed, lastperm, actualsize, i;

   BX_TRY_BEGIN
   {

      /* Open the communication session to testcard on serial port
         COM1 */

      BX_TRY(BestXOpen(&handle, BX_PORT_RS232, BX_PORT_COM1));

      /* Set the baud rate for the serial port */

      BX_TRY(BestXRS232BaudRateSet(handle, BX_BD_57600));

      /* Set up a block transfer for the testcard. */

      /* ... */

      /* Initialize PPR */

      BX_TRY(BestXPprInit(handle));
      BX_TRY(BestXPprGenDefaultSet(handle));

      /* Set all RI ppr properties to their defaults */
      BX_TRY(BestXPprRIDefaultSet(handle));

      /* Set PPR generics for report handling, etc. */

      BX_TRY(BestXStatusRead(handle, BX_STAT_BUSWIDTH, &width));
      BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_BUSWIDTH, width));
      BX_TRY(BestXStatusRead(handle, BX_STAT_BUSSPEED, &speed));
      BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_BUSSPEED, speed));
      BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_XFERCLKS, 5));
```

```
/* THIS IS JUST AN ESTIMATE */

BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_ALGORITHM,
BXPPR_GEN_ALGORITHM_PERM));

BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_LEVEL,
BXPPR_GEN_LEVEL_DATA));

BX_TRY(BestXPprGenSet(handle, BXPPR_GEN_ADBLIMITATION , 1));

BX_TRY(BestXPprRIBlkPermSet(handle,
BXPPR_RIBLKPERM_DIRECTION, BXPPR_RIBLKPERM_DIRECTION_READ));

BX_TRY(BestXPprRIBlkPermSet(handle,
BXPPR_RIBLKPERM_BLOCKSIZE, 0x100000)); // a 1M block

BX_TRY(BestXPprRIBlkPermSet(handle,
BXPPR_RIBLKPERM_BUSADDR_LO, 0x20000000));

BX_TRY(BestXPprRIBlkPermSet(handle,
BXPPR_RIBLKPERM_BUSADDR_HI, 0x30000000));

BX_TRY(BestXPprRIBlkPermSet(handle, BXPPR_RIBLKPERM_INTADDR,
0x0000));

BX_TRY(BestXPprRIBlkPermSet(handle, BXPPR_RIBLKPERM_FILLGAPS,
BX_YES)); // this is the default

/* Start filling the Block transfer memory at line 0 of 256
*/

BX_TRY(BestXPprRIBlkPermSet(handle,
      BXPPR_RIBLKPERM_STARTOFFSET, 0)); //this is the default

/* Define the permutation lists.  These lists of values will
be permutated against each other */

/* The permutated BUSCMD values are dependent on
/* BXPPR_RIBLKPERM_DIRECTION * /

BX_TRY(BestXPprRIBlkListSet(handle, BXPPR_RIBLK_BUSCMD, \
"BX_RIBLK_BUSCMD_MEM_READDWORD,BX_RIBLK_BUSCMD_MEM_READBLOCK,
BX_RIBLK_BUSCMD_MEM_WRITE, BX_RIBLK_BUSCMD_MEM_WRITEBLOCK"));

BX_TRY(BestXPprRIBlkListSet(handle, BXPPR_RIBLK_NOSNOOP,
                            "0,1"));

BX_TRY(BestXPprRIBlkListSet(handle, BXPPR_RIBLK_ALIGN,
                            "0,1,2,3,4,5,6,7"));

BX_TRY(BestXPprRIBlkListSet(handle, BXPPR_RIBLK_BYTEN,
                            "0,1,2,3,4,5,6,7,8,9,
                            10,11,12,13,14,15"));

BX_TRY(BestXPprRIBlkListSet(handle, BXPPR_RIBLK_NUMBYTES,
                            "1,2,3,4,5,6,7,8,16,32,64,
                            128,256,512,1024,4096"));

BX_TRY(BestXPprRIBehPermDefaultSet(handle));

BX_TRY(BestXPprRIBehListSet(handle, BX_RIBEH_BYTECOUNT,
                            "33, 64, 72, 128, 4096"));
```

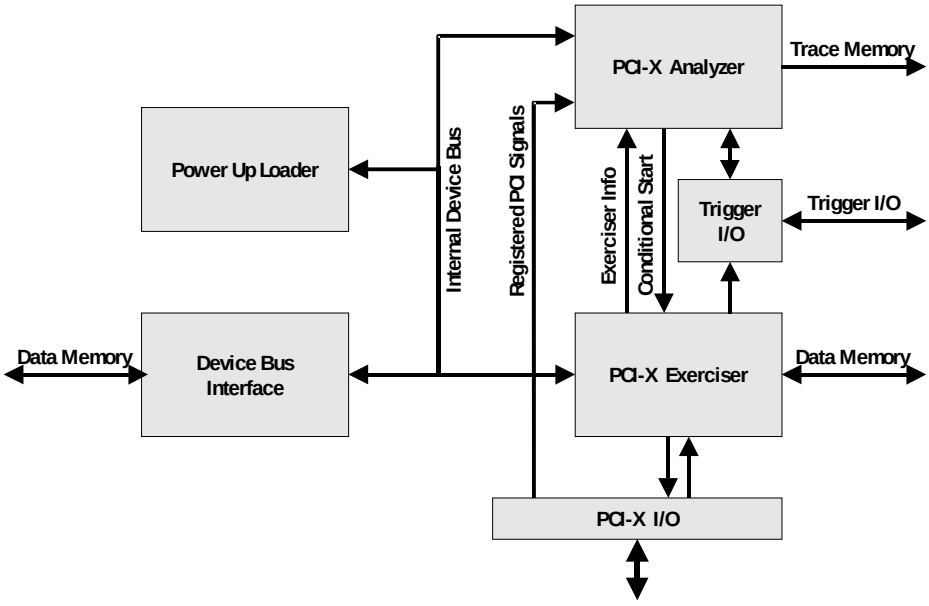
```
BX_TRY(BestXPprRIBehListSet(handle, BX_RIBEH_REQ64, "0,1"));
BX_TRY(BestXPprRIBehListSet(handle, BX_RIBEH_QUEUE,
                    "BX_RIBEH_QUEUE_A, BX_RIBEH_QUEUE_B"));

BX_TRY(BestXPprRIBehListSet(handle, BX_RIBEH_DISCONNECT,
                           "1,2,3,4,5,6,7"));

BX_TRY(BestXPprRIBehListSet(handle, BX_RIBEH_DELAY,
                           "100,200,300"));

BX_TRY(BestXPprRIBehPermSet(handle, BXPPR_BEHPERM_FIRSTPERM,
                              1));

/* Sets report properties: Include C-API abbreviation in the
report that should have a length of 50 lines. */

BX_TRY(BestXPprReportSet(handle, BXPPR_REPORT_CAPI, BX_YES));

BX_TRY(BestXPprReportSet(handle, BXPPR_REPORT_CONTENTS, 50));

/* Since we know we are creating a very large permutation
list, we will need to iterate */

/* Several times to complete all permutations. */
/* How many permutations are we doing? */

BX_TRY(BestXPprRIBlkResultGet(handle,BXPPR_RIBLKRES_LASTPERM,
                              &lastperm));

BX_TRY(BestXPprRIBlkResultGet(handle,
                              BXPPR_RIBLKRES_ACTUALSIZE,
                              &actualsize));

printf("lastperm = %ld actualsize = %d\n", lastperm,
        actualsize);

for (i=1; i < lastperm; i+=actualsize)
{
   char num[6];
   char reportname[80] = "c:\\temp\\PprReport";

   /* Remember that the block transfer memory can only hold
   256 entries.  We are creating a much larger permutation
   set, so we must iterate this to run all different
   combinations */

   BX_TRY(BestXPprRIBlkPermSet(handle,
                              BXPPR_RIBLKPERM_FIRSTPERM, i));

   BX_TRY(BestXPprRIBlkResultGet(handle,
                                 BXPPR_RIBLKRES_ACTUALSIZE,
                                 &actualsize));

   /* Program the testcard */

   BX_TRY(BestXPprProg(handle));

   /* This is required to program the card */

   BX_TRY(BestXExerciserProg(handle));
```

```
            /* Create a report file name */
            strcat (reportname, _itoa(i, num, 10));
            strcat (reportname, ".txt");
            /* Generate a report file for each new permutation
            (iteration */
            BX_TRY(BestXPprReportFile(handle, reportname));
            time( &start );
            BX_TRY(BestXExerciserRun(handle)); //start the exerciser …
            time( &finish );
         //…and run it for EXECUTION_TIME seconds per permutation list
            while ((difftime(finish,start)) < EXECUTION_TIME)
               time(&finish);
            BX_TRY(BestXExerciserStop(handle));
         } // end of for loop
      BX_TRY(BestXClose(handle));
      }
      BX_TRY_CATCH
      {
         // cleanup, if necessary
         printf("%s\n", BestXErrorStringGet(BX_TRY_RET));
      }
      return 0;
}
```

Synchronizing the Environment

The Agilent E2929A/B testcard provides application interfaces for exchanging information between the testcard and the test environment during the run time of the test application.

The following figure shows the components of the PCI-X ASIC and the interfaces to the environment.



The following sections provide information about synchonizing to the environment.

- *"Card Status Reporting" on page 191* gives information about using the testcard's status register.

  This information is useful for evaluating test results or for debugging and evaluating errors.

- *"Generic Testcard Setup and Power-Up Control" on page 194* shows how to control the testcard's power-up and reset behavior.

  This information is useful for tests focusing on the power-up behavior of the system under test. It is also for when the testcard hangs and you need to unlock it.

- *"Programming the Mailbox" on page 195* shows how data can be exchanged between applications running on the control PC and the system under test.
- *"Programming the Trigger I/O" on page 199* shows how to use the trigger input and output lines.
- *"Programming the Display" on page 203* shows how to write data to the 7-segment display.

Card Status Reporting

The testcard status register can be used, for example, to evaluate the test result after the test run, and to debug and evaluate errors.

You can read the card status register for the following information:

- Data compare errors
- System and parity errors
- Initiator and target aborts
- Asserted interrupts
- Information about the test status
- The state of DEVSEL# (bit 0), STOP# (bit 1), and TRDY# (bit 2) during the rising edge of RST#
- The piggyback ID
- The current state of the trigger I/O pins
- The state of the hardware dip switch
- The bus mode (PCI or PCI-X)
- The setting of the 'invisible' hardware jumper on the board
- Bus width and bus speed
- The current status of the PCI/PCI-X bus reset signal
- Information about whether the PCI-X bus has been reset since the last check

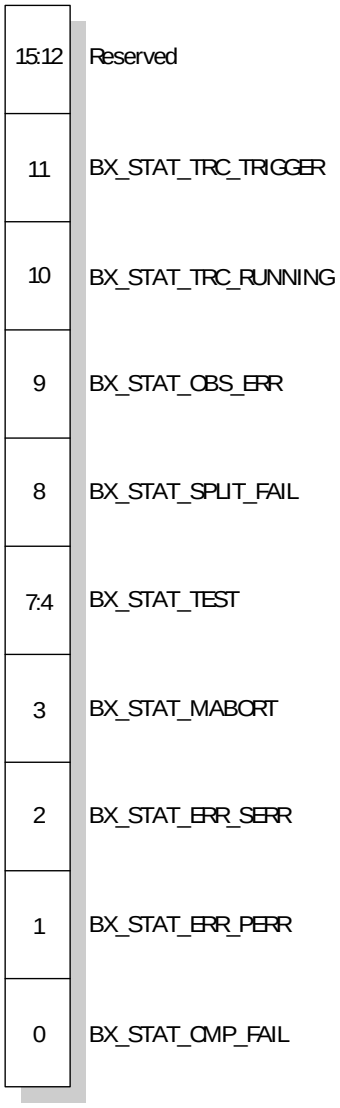
For details, please refer to "bx_statustype" in the *Agilent E2929A/B Opt. 320 C-API/PPR Programming Reference*.

How to Access the Card Status Register

Reading the Testcard Status

To read the testcard status, you can use the C-API call *BestXStatusRead*.

The following figure shows the testcard status register layout (offset `0x52`):



Programming Steps

The testcard status register can be accessed as follows:

1. Read a specific bit of the testcard status register with *BestXStatusRead*.
2. Clear a specific bit of the testcard status register with *BestXStatusClear* to ensure a specific register condition.

NOTE Only some of the status register properties can be cleared. For details, please refer to "bx_statustype" in the *Agilent E2929A/B Opt. 320 C-API/PPR Programming Reference*.

Example for Accessing the Card Status Register

Task Poll the status register by using the C-API to:

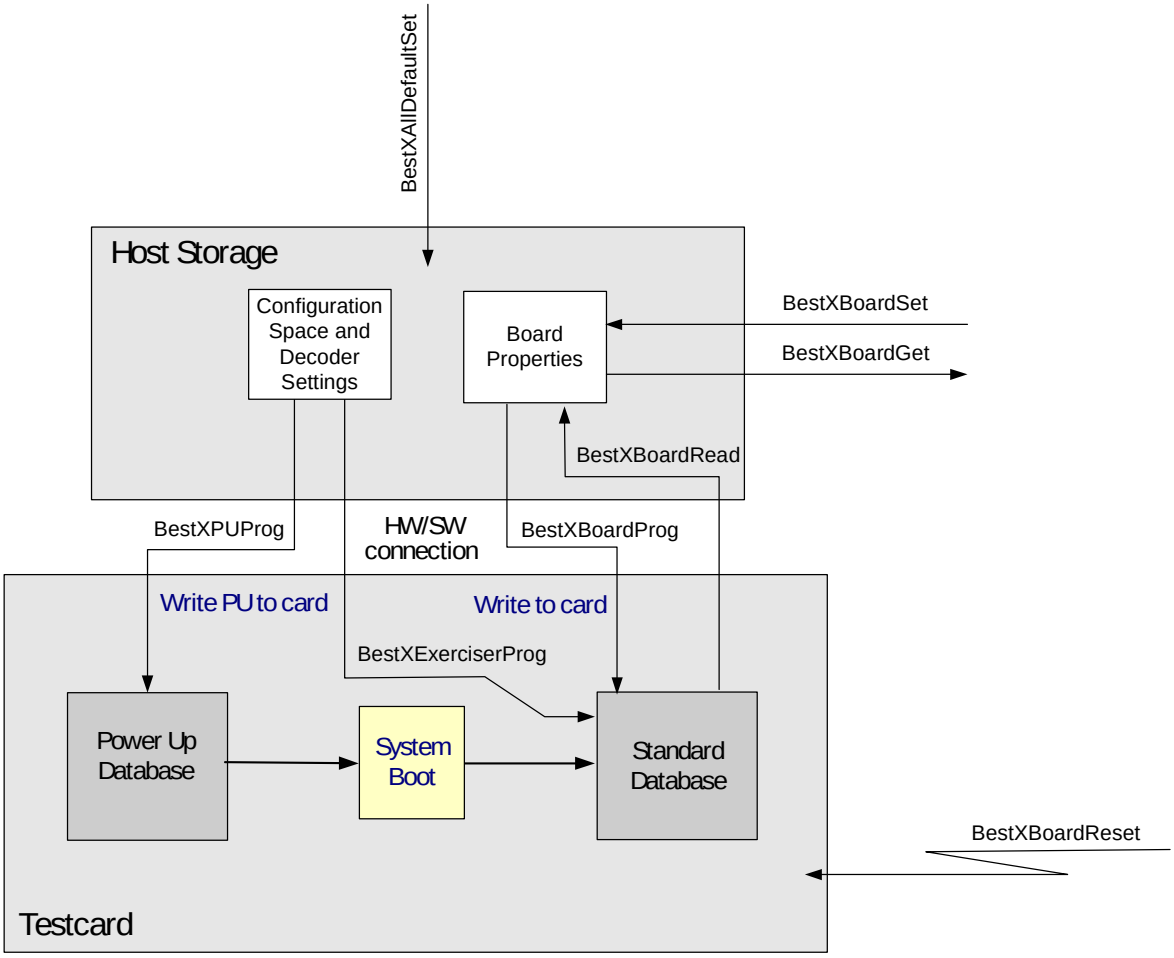
- Detect whether the Analyzer is running.
- Query and clear interrupts.

Implementation

```
/* Detect whether the Analyzer is running */
BX_TRY(BestXStatusRead(handle, BX_STAT_TRC_RUNNING, &trcstat));
if (trcstat)
   printf("Analyzer running\n");
/* Query the interrupt */
BX_TRY(BestXStatusRead(handle, BX_STAT_INTB, &val));
printf("the value of the number.. before %d\n",val);
/* Clear the interrupt */
BX_TRY(BestXStatusClear(handle, BX_STAT_INTB));
BX_TRY(BestXStatusRead(handle, BX_STAT_INTB, &val));
printf("the value of the number.. after  %d\n", val);
```

Generic Testcard Setup and Power-Up Control

The following figure shows all functions used to program the power-up and reset behavior of the testcard. The figure also displays all memories controlled by these functions.



How to Program Generic Testcard Properties and Power-Up Control

Programming Options

The power-up control of the Exerciser and Analyzer testcard allows the following options:

- Setting properties to the host
- Setting properties to the testcard

Setting Properties to the Host

To set properties on the host, you can:

- Set all properties on the host to default values with *BestXAllDefaultSet*. No direct testcard access is done.
- Set board properties with *BestXBoardDefaultSet* and *BestXBoardSet*.

Setting Properties to the Testcard

To set properties to the testcard, you can:

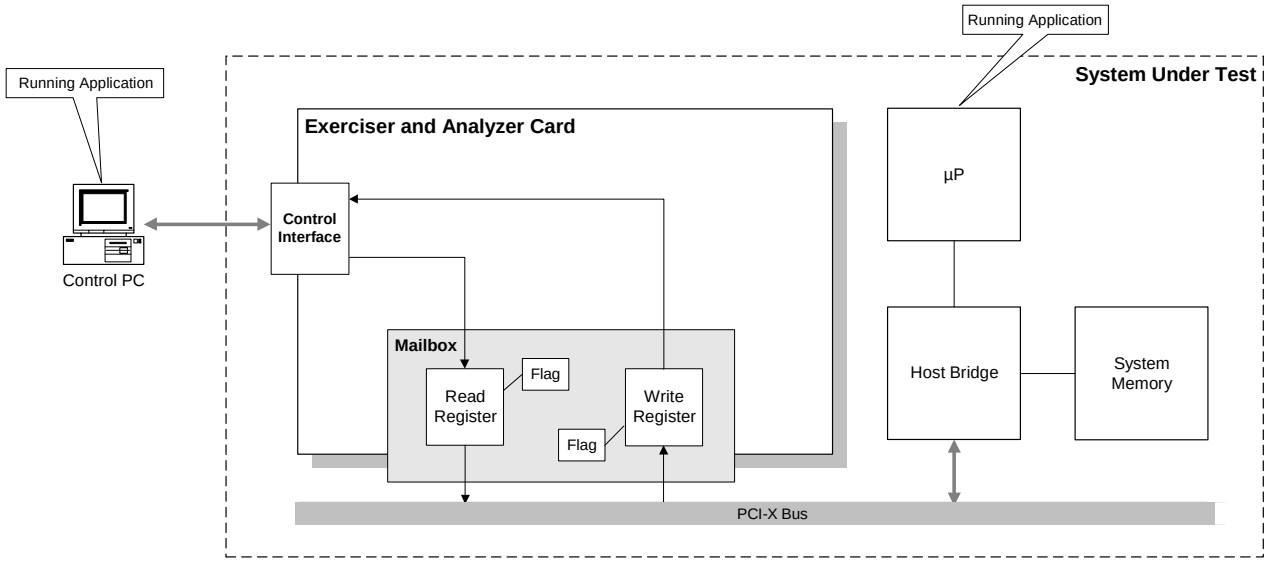
- Store the current settings of the configuration space including decoder settings to the power up database with *BestXPUProg*. With the next power-up the testcard is initialized with these settings.
- Store the non-volatile configuration space content into the testcard with *BestXExerciserProg*.
- Use the current board settings on the host storage as current settings by loading them to the standard database with *BestXBoardProg*. To read settings, use *BestXBoardRead*.
- To issue a board reset, use *BestXBoardReset*.

Programming the Mailbox

The mailbox of the Agilent E2929A/B testcard allows communication between a program running on the system under test and a program running on an external control PC. Communication to the control PC is done via either the control interface, RS-232 or the Fast Host Interface (the PCI bus can also be used as the control interface if the control PC is simultaneously the system under test).

The mailbox consists of two 32-bit registers, one Read register, and one Write register. This enables full duplex operation. Each register is equipped with a flag that is set when data is written into the register, and reset if the register is read.

The figure below shows the principle of the mailbox:



The mailbox can be accessed by:

- Functions provided by the C-API
- Direct PCI-X access, that is, by a programmable address range in memory space, or I/O space, or configuration space

The flags are held in the *mailbox status register*.

Access by Functions

The mailbox can be accessed by using the mailbox functions either from the control PC or from the system under test.

Direct PCI-X Access to the Mailbox

The mailbox registers are located in the private section of the Agilent E2929A/B testcard's configuration space. They can be read or written by using configuration commands. The mailbox register addresses are shown in the table below.

An access to the lowest byte of each register generates an interrupt that can be used to inform the communication partner about the access. If this is used, the lowest byte should be accessed either simultaneously with or directly after access to the other bytes.

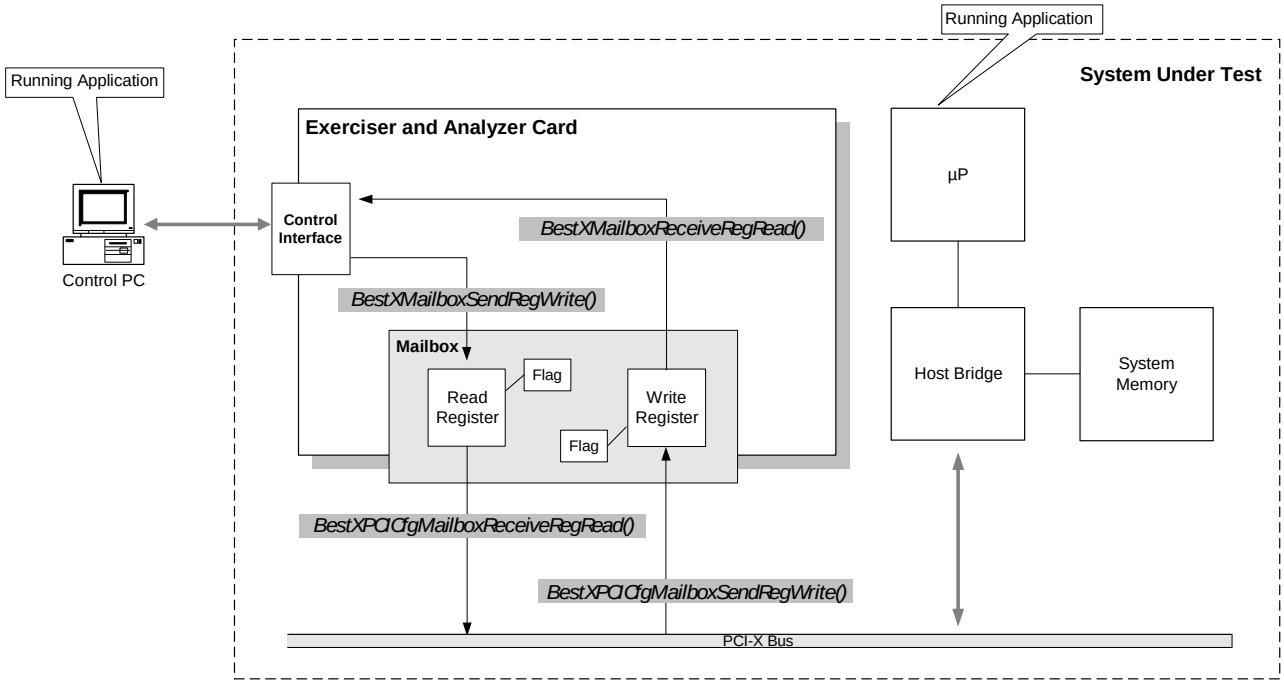
Mailbox Status Register

The following table shows the mailbox status register in the configuration space:

Offset Config	Bits	Type	Operation	Meaning
4C\h	[31:0]	RW	Conf. Read	Reads the mailbox.
			Conf. Write	Writes to the mailbox.
50\h	0	RO	Conf. Read	Flag of the mailbox **write** register: 0 = mailbox empty, write possible 1 = don't write, mailbox contains data
	1	RW	Conf. Read	Flag of the mailbox **read** register: 0 = mailbox is empty 1 = mailbox contains data **Note:** If you read the mailbox via PCI, reset this flag by writing a 1 to this bit.
			Conf. Write	Generates an interrupt for the on-board CPU. This informs the CPU that the mailbox register has been read and clears the flags.

How to Program the Mailbox

The following figure shows the available mailbox functions and the application:



Programming Steps for Access via PCI-X Bus

To access the mailbox via PCI-X bus, the following steps are required:

1. Identify the testcard.

   Use *BestXDevIdentifierGet*.

   Because multiple PCI-X testcards can be plugged into the system under test, the Agilent E2929A/B testcard needs to be identified for mailbox access.

2. To write data to the mailbox via the PCI-X bus, use *BestXPCICfgMailboxSendRegWrite*.

   This function automatically checks the status flag. Unread data will not be overwritten.

   If the mailbox contains unread data, first read the data to reset the flag with *BestXPCICfgMailboxReceiveRegRead*.

Programming Steps for Access via Control PC

To access the mailbox via the control PC:

- Send and receive data using the control interface.

  Use *BestXMailboxSendRegWrite* and *BestXMailboxReceiveRegRead* respectively.

Example for Programming the Mailbox

The following code fragments give examples of the two ways of accessing the mailbox:

Task Write data to the mailbox via the PCI-X bus.

Implementation

```
/* Identify the testcard and write data to the mailbox until the
flag indicates that data has been written successfully. */
BX_TRY(BestXDevIdentifierGet(0x103C, 0x2929, 0, &devid));

do {
   BX_TRY(BestPCICfgMailboxSendRegWrite(devid, data, &status));
} while(status == 0);
```

Task Read data from the mailbox via the control PC.

Implementation

```
/* Read from the mailbox until valid data can be read from the
mailbox. If the status bit is set, previously unread "mail" is
returned as the value. */

do {
   BX_TRY(BestXMailboxReceiveRegRead(handle, &data, &status));
} while(status == 0);
```

Programming the Trigger I/O

The Agilent E2929A/B testcard provides four trigger-in and trigger-out signals.

Trigger-In The trigger-in pins are always enabled.

They can be used to:

- Synchronize the traffic generation of the testcard on an external trigger event.
- Synchronize multiple testcards among themselves.

Trigger-Out

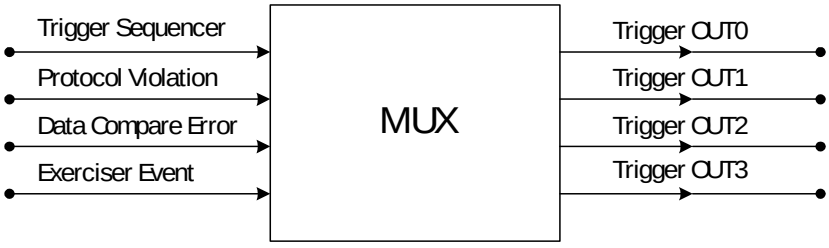
The trigger-out pins need to be enabled before they can be used. Each trigger-out line can be programmed as as either input, open-drain output, or totem-pole output.

The trigger-out pins can be used to:

- Trigger an external ocilloscope or logic analyzer on the following events:
  - On a data miscompare
  - On a protocol violation
  - When the built-in analyzer triggers,
  - On a particular exerciser event
- Trigger other testcards on the involvement of this testcard in a particular transaction.

Trigger I/O Configuration

The input and output pins are configured as follows:



To set up the trigger I/O, you have to enable the outputs and select the trigger output source.

How to Program the Trigger I/O

Programming Steps

To program the trigger I/O:

1 Enable the trigger output(s) to be used. Use *BestXBoardSet* and set the BX_BOARD_TRIGIO<n>_MODE property to the required value (open-drain or totem-pole output), where <n> means the trigger output(s) 0 … 3.

NOTE To disable a specific trigger output, set the respecitive BX_BOARD_TRIGIO<n>_MODE property to input.

2 After you have enabled the trigger output(s), select the trigger output source.

Use *BestXBoardSet* to map the trigger-out signal(s) to internal signal(s) (trigger sequencer signal, protocol error, data compare error or exerciser event) by setting the BX_BOARD_TRIGIO<n>_OUT property.

By default, the input/output pins are mapped as follows:

Trigger Serquencer	->	Out0
Protocol Error	->	Out1
Data Compare Error	->	Out2
Exerciser Event	->	Out3

3 Write the board properties to the testcard with *BestXBoardProg*.

4 If you selected an exerciser event as trigger output source:

- Specify the event and the exact location within this event.

  Use *BestXExerciserGenSet* and set the generic exerciser properties BX_EGEN_TRIG_SOURCE and BX_EGEN_TRIG_NUM to the required values.

- Write the exerciser generic properties to the testcard with *BestXExerciserProg*.

5 Depending on the selected source, run the trace memory or the exerciser of the testcard, or both:

- If you selected an exerciser event and/or a data compare error as output source, use *BestXExerciserRun*.
- If you selected a trigger sequencer signal and/or a protocol error as output source, use *BestXTraceRun*.

Example for Programming the Trigger I/O

Task Set up the E2929A so that it generates a trigger output on the fourth pin during the second block transfer.

Implementation

```
##include <xpciapi.h>

int main(int argc, char* argv[])
{
   BX_TRY_VARS_NO_PROG;

   /* additional local variable declarations, here */
   bx_handletype handle;

   BX_TRY_BEGIN
   {
      BX_TRY(BestXOpen(&handle, BX_PORT_RS232, BX_PORT_COM1));
      BX_TRY(BestXRS232BaudRateSet(handle, BX_BD_57600));
      BX_TRY(SetupForTriggerIO(handle));

      /* enable trigger out 3 and set the output to totem pole */

      BX_TRY(BestXBoardSet(handle, BX_BOARD_TRIGIO3_MODE,
            BX_BOARD_TRIGIO_MODE_TOTEMPOLE));

      /* connect an exerciser event to trigger out 3 */

      BX_TRY(BestXBoardSet(handle, BX_BOARD_TRIGIO3_OUT,
            BX_BOARD_TRIGIO_OUT_TRIGSOURCE));

      BX_TRY(BestXBoardProg(handle));

      /* specify that the trigger output is asserted at the start
         of the second requester-initiator block */

      BX_TRY(BestXExerciserGenSet(handle, BX_EGEN_TRIG_SOURCE,
            BX_EGEN_TRIG_SOURCE_RIBLK));

      BX_TRY(BestXExerciserGenSet(handle, BX_EGEN_TRIG_NUM, 1));

      BX_TRY(BestXExerciserProg(handle));
      BX_TRY(BestXExerciserRun(handle));
      BX_TRY(BestXClose(handle));

   }

   BX_TRY_CATCH
   {
      // cleanup, if necessary
      printf("%s\n", BestXErrorStringGet(BX_TRY_RET));
   }

   BX_ERRETURN(BX_TRY_RET);

}
```

Programming the Display

Programming Steps

To write a value or a string to the 7-segment display, the following steps are required:

1. Before writing values to the display, ensure that the LED display mode is set to “user mode”.

   To verify this mode, read the board property `BX_BOARD_DISPLAY` with *BestXBoardGet*. If `BX_BOARD_DISPLAY_USER` is set, proceed with step two. Otherwise, set this property value with *BestXBoardSet*.

2. To write a value to the LED display, use *BestXDisplayWrite*. To write a string to the 7-segment display, use *BestXDisplayStringWrite*.

Example for Programming the Display

Task Write “Hello World” to the 7-segment display.

Implementation

```
#include "stdafx.h"
#include "xpciapi.h"

int main(int argc, char* argv[])
{
   BX_TRY_VARS_NO_PROG;

   /* Enter additional local variable declarations here */
   bx_handletype handle;

   BX_TRY_BEGIN
{
   /* Open the communication session to testcard, initialize  */
   /* internal structures */

   BX_TRY(BestXOpen(&handle, BX_PORT_RS232, 1));

   /* If using RS232, set baud rate: */
   BX_TRY(BestXRS232BaudRateSet(handle, BX_BD_57600));

   /* Insert here your C-API calls */
   /* For example:*/
   /* Write “Hello World” to the display.*/
   int i;

   for (i=0;i<10;i++)

   {
      BX_TRY (BestXDisplayStringWrite(handle, "HEL-"));
      BX_TRY (BestXDisplayStringWrite(handle, "HEL\\"));
      BX_TRY (BestXDisplayStringWrite(handle, "HEL|"));
      BX_TRY (BestXDisplayStringWrite(handle, "HEL/"));
   }

   /* Close the session to deallocate memory. */
      BX_TRY(BestXClose(handle));
   }

   BX_TRY_CATCH
   {
       printf(BestXErrorStringGet(BX_TRY_RET));
      /* cleanup, if necessary */
   }
   return 0;

}
```

Index

A

Accumulated Error Register 91
Algorithms
  Permutation 131
Alignment 136
Application Interfaces
  Programming 189
Arbitration Algorithm
  Automatical 68
  Constant 69
  Incremental 70
  Random 71
Automatical Arbitration
  Arbitration Algorithm 68

B

Basic Terms 123
Behavior Memory
  Completer-Target 48, 57
  Requester-Initiator 35
Behavior of Block Transfers
  Example 39
  Programming 35
  Programming Steps 38
Benefits
  PPR Software 21
Block 134
  Permutation Properties 134
  Variation Parameters 136
Block Permutation Properties
  Block Permutation Properties 134
  Bus Address 135
  Compound Block Size (CBS) 135
  Fill Gaps 135
  First Permutation Number 135
  Internal Address 135
  Resource 135
  Size Limit 136
  Start Offset 136
  Tuples 136
Block Permutations
  Report Section 160
Block Transfers
  Example 32
  Programming Steps 31
Bus Address
  Block Permutation Properties 135
Bus Commands 136
Byte Enable Variation 173
Byte Enables 137

C

C Programming Libraries 13
Calculations of Coverage 138
C-API
  Generic Functionality 14
Card Status
  Reporting 191
Card Status Register
  Contents 191
  Example 193
Card Status Register Access
  Programming Steps 192
CBS (= Compound Block Size) 135
Completer-Initiator Behavior
  Example 60
  Programming 57
  Programming Steps 59
Completer-Initiator Behavior Permutations
  Programming 150
Completer-Initiator Behaviors
  Memory Design 50, 59
Completer-Initiator Device
  Programming the Exerciser 55
Completer-Target Behavior
  Example 50
  Programming 48
  Programming Steps 49
Completer-Target Behavior Memory 48, 57
Completer-Target Behavior Permutations
  Programming Steps 148
Completer-Target Device
  Programming the Exerciser 40
Components
  PCI-X ASIC 189
  Performance Sequencer 111
  Trigger Sequencer 96
Compound Block 134
Compound Block Size
  Block Permutation Properties 135
Config Read Access 46
Config Write Access 46
Configuration
  Trigger I/O 200
Configuration Space
  Programming 45
  Setting the Register Mask 45
  Setting the Register Value 45
Connection
  Programming Steps 18
Constant Arbitration
  Arbitration Algorithm 69
Contributions
  of the PPR Software 120
Controlled Protocol Corner Cases
  Creating 21
Counter
  Performance 111
Coverage 125
  Calculations 138
  Completer-Initiator Behavior 150
  Completer-Target Behavior 147
  Requester-Initiator Behavior 143
  Requester-Initiator Block Permutations 137
  Requester-Target Behavior 153
  Uncovered Permutations 173
Creating Controlled Protocol Corner Cases
  Benefit 21

D

Data Alignment
  Data Memory 81
Data Generator
  Bit Assignment 74
  Example 75
  Features 73
  Programming Steps 75
  Programming,Programming Data Generator 73
  Properties 73
Data Memory
  Data Alignment 81
  Example 83
  Programming 81
  Programming Steps 83
Data-Integrity Testing
  Benefit 21
Decoder
  Example 44
  Programming Steps 43
Decoders Linked to the Configuration Space Header 42
Detailed Report
  Benefit 22
Deterministic and Reproducible Tests
  Benefit 22
Directory Structure 13
Disconnect (Requester-Initiator Behavior) 36
Documentation Overview 9
Downloading Settings to the Testcard
  Exerciser Programming 27

E

Emulating Typical Peripheral Traffic
  Benefit 21
Environment
  Synchronizing 189
Error Checking
  Handle-Based 16
  Non-Handle-Based 16
Error Register
  Contents 90
  Design 91
Errors Injection
  Example 80
  Programming 76
  Programming Steps 78
Example
  Behavior of Block Transfers 39
  Block Transfers 32
  Card Status Register 193
  Completer-Initiator Behavior 60
  Completer-Target Behavior 50
  Data Generator 75
  Data Memory 83
  Decoder 44
  Errors Injection 80
  Generating PPR Reports 157
  Generic Requester-Initiator Properties 30
  Generic Requester-Target Properties 63
  Modifying the Configuration Space 47
  Pattern Terms 95
  PCI-X Interrupts 87
  PPR Administration 133
  PPR Test Setup 133
  Programming CI Behavior Permutations 152
  Programming CT Behavior Permutations 149
  Programming Generic Completer-Target Properties 53
  Programming Requester-Initiator Block Permutations 141
  Programming Requester-Target Behavior Permutations 155
  Programming the Mailbox 199
  Programming the Performance Sequencer 115
  Protocol Observer 93
  Requester-Target Behavior 65
  Running a PPR Test 157
  Scheduling Block Transfers 72
  Scheduling Split Completions 72
  Split Completion Decoder 63
  Split Condition 54
  Test Design 128
  Trace Memory 106
  Trigger Sequencer 100
Example for Generating Sequences 35
Exerciser
  Downloading Settings to the Testcard 27
  Running 27
Expansion ROM
  Programming 80

F

Fast Host Interface Port
  Initialization 18
Feedback Counter
  Enable Condition 97
  Preload Condition 98
  Trigger Sequencer 99
Fill Gaps
  Block Permutation Properties 135
First Error Register 91
First Permutation Number
  Block Permutation Properties 135
Fitting List
  Master Block 164
Functional Test Phase 118
Functionality
  C-API 14
  PPR 15

G

Gaps 135
General PPR Properties
  Report Section 159
Generating
  Permutations 123
Generating PPR Reports
  Example 157
Generating Sequences 35
  Example 35
Generic Completer-Initiator Properties
  Programming 55
Generic Completer-Target Properties
  Programming 52
  Programming Steps 52
Generic Requester-Initiator Properties
  Example 30
  Programming 29
  Programming Steps 29
Generic Requester-Target Properties
  Example 63
  Programming 61

H

Handle Initialization 17
Handle-Based Error Checking 16

I

Incremental Arbitration
  Arbitration Algorithm 70
Initialization
  Fast Host Interface Port 18
  PCI-X Port 17
  Programming Steps 18
  RS-232 Serial Interface 17
  USB Port 17
Internal Address
  Block Permutation Properties 135
Interrupt
  Status Register 86
Interrupt Status Register 86

L

Libraries
  for C Programming 13

M

Mailbox
  Status Register 197
Mailbox Access via Control PC 198
Mailbox Access via PCI-X Bus 198
Mask
  of Rules 90
Master Block
  Fitting List 164
  Permutation Table (Example) 162
Memory Design
  Completer-Initiator Behaviors 59
  Completer-Target Behaviors 50
  Requester-Intitiator Behaviors 38
  Requester-Intitiator Block Transfers 32
  Requester-Target Behaviors 65
Modifying the Configuration Space
  Example 47
  Programming Steps 45

N

Next State 97
No Snoop 137
Non-Handle-Based Error Checking 16

O

Operation Principles 121
Optimizing Testing Time 172
Overview
  Documentation 9

P

Parameters 123
Pattern Terms
  Example 95
  Programming 94
  Programming Steps 94
  Trigger Sequencer 100
  Using 94

PCI-X ASIC
Components 189
PCI-X Exerciser
Programmable Components 23
Programming Concept 23
PCI-X Interrupts
Example 87
PCI-X Port
Initialization 17
PCI-X Protocol Behavior Permutations within Programmable Constraints
Benefit 22
PCI-X Protocol Permutation and Randomizer
Functionality 15
Performance Measurement 114
Programming Steps 114
Performance Sequencer
Components 111
Programming 111
Performance Sequencer Memory
Programming Model 113
Performing Parameter Permutations 122
PERM 131
Permutating Algorithm 131
Permutation Results
Requester-Initiator Report Section 167
Permutation Table 124
Permutations 123
Generating 123
Platform-Dependence 14
Power-Up Behavior
Programming 194
Power-Up Control
Programming Options 195
PPR
Background Information 117
Functionality 15
PPR Administration 131
Example 133
Programming Steps 132
PPR Software
Benefits 21
Contributions 120
PPR Test
Running 157
PPR Test Run
Programming Steps 157
PPR Test Setup 131
Example 133
Programming Steps 132
Predictable Testing Time
Benefit 22
Presetting Values 172
Program Header 127
Programmable Behaviors Regarding Decoders 44
Programmable Components
PCI-X Exerciser 23
Programmable Constraints
for PCI-X Protocol Behavior Permutations 22
Programmable Decoder Properties 43
Programmable Memories 121
Programming
Application Interfaces 189
Behavior of Block Transfers 35
Completer-Initiator Behavior 57
Completer-Initiator Behavior Permutations 150
Completer-Target Behavior 48
Configuration Space 45
Data Memory 81
Errors Injection 76
Expansion ROM 80
Generic Completer-Initiator Properties 55
Generic Completer-Target Properties 52
Generic Requester-Initiator Properties 29
Generic Requester-Target Properties 61
Pattern Terms 94
Performance Sequencer 111
Power-Up Behavior 194
PPR Administration 131
PPR Test Setup 131
Requester-Initiator Behavior Permutations 142
Requester-Initiator Block Transfers 30
Requester-Target Behavior 63
Requester-Target Behavior Permutations 153
Reset Behavior 194
Split Completion Decoder 63
Split Condition 53
Target Decoder 41
Trace Memory 104
Trigger I/O 199
Trigger Sequencer 96
Programming CI Behavior Permutations
Example 152
Programming CT Behavior Permutations
Example 149
Programming Data Transfer
Data Transfer
Programming 28
Programming Generic Completer-Target Properties
Example 53
Programming Interfaces 12
Programming Model
Performance Sequencer 113
Programming Options
Power-Up Control 195
Reset Control 195
Programming Requester-Initiator Block Permutations 141
Programming Requester-Target Behavior Permutations 155
Programming Steps 114
Behavior of Block Transfers 38
Block Transfers 31
Card Status Register Access 192
Completer-Initiator Behavior 59
Completer-Target Behavior 49
Completer-Target Behavior Permutations 148
Connection 18
Data Generator 75
Data Memory 83
Decoder 43
Errors Injection 78
Generic Completer-Target Properties 52
Generic Requester-Initiator Properties 29
Initialization 18
Mailbox Access via Control PC 198
Mailbox Access via PCI-X Bus 198
Modifying the Configuration Space 45
Pattern Terms 94
PPR Administration 132
PPR Report Generation 156
PPR Test Run 157
PPR Test Setup 132
Protocol Observer 91
Requester-Initiator Block Permutations 140
Requester-Target Behavior 64
Requester-Target Behavior Permutations 154
Scheduling Block Transfers 72
Scheduling Split Completions 72
Sequencer 99
Setting Power-Up and Testcard Properties to the Host 195
Setting Power-Up and Testcard Properties to the Testcard 195
Split Condition 54
Trace Memory 105
Trigger Sequencer 99
Writing a C Program 127
Programming the Exerciser
as Completer-Initiator Device 55
as Completer-Target Device 40
as Requester-Initiator Device 28
as Requester-Target Device 61
Programming the Mailbox
Example 199
Programming the Performance Sequencer
Example 115
Protocol Observer
Example 93
Programming Steps 91

R

RAND 131
Random Arbitration
Arbitration Algorithm 71

Randomizing Algorithm 131
Reading the Testcard Status
  Options 192
Relaxed Order 137
Repetition Length 125
Report
  Analyzing 159
  Listing 174
Report Generation (PPR)
  Programming Steps 156
Report Properties
  Report Section 160
Report Sections
  Block Permutation Results 161
  Block Permutations 160
  General PPR Properties 159
  Header 159
  Master Block Permutation 160
  Report Properties 160
  Requester-Initiator Behavior Permutation 167
Reporting
  Card Status 191
Requester-Initiator Behavior Groups (PPR) 142
Requester-Initiator Behavior Memory 35
Requester-Initiator Behavior Permutations
  Programming 142
Requester-Initiator Block Permutations
  Programming 134
  Programming Steps 140
Requester-Initiator Block Transfers
  Programming 30
Requester-Initiator Device
  Programming the Exerciser 28
Requester-Intitiator Behaviors
  Block Transfers 32
  Memory Design 38
Requester-Target Behavior
  Example 65
  Programming 63
  Programming Steps 64
Requester-Target Behavior Groups (PPR) 153
Requester-Target Behavior Permutations
  Programming 153
  Programming Steps 154
Requester-Target Behaviors
  Memory Design 65
Requester-Target Device
  Programming the Exerciser 61
Reset Behavior
  Programming 194
Reset Control
  Programming Options 195
Resource
  Block Permutation Properties 135
RS-232 Serial Interface
  Initialization 17
Rule Mask 90
Running
  PPR Test 157
Running a PPR Test
  Example 157
Running the Exerciser 27

S

Scheduling Block Transfers
  Example 72
  Programming Steps 72
Scheduling Split Completions
  Example 72
  Programming Steps 72
Sequencer
  Programming Steps 99
  Set Up 97
Sequences
  Generating 35
Setting Power-Up and Testcard Properties to the Host
  Programming Steps 195
Setting Power-Up and Testcard Properties to the Testcard
  Programming Steps 195
Setting the Configuration Space Register Mask 45
Setting the Configuration Space Register Value 45
Size Limit
  Block Permutation Properties 136
Split Completion Decoder
  Example 63
  Programming 63
Split Condition
  Example 54
  Programming 53
  Programming Steps 54
Start Address Alignment 136
Start Offset
  Block Permutation Properties 136
State 97
Status Register
  Interrupt 86
  Interrupts 86
  Mailbox 197
Storage Qualifier 104
  Condition 98
Storing and Analyzing Bus Traffic
  Benefit 22
Stressing on Multiple PCI/PCI-X Buses
  Benefit 22
Synchronizing
  Environment 189
System Assurance 119
System Integration 119

T

Target Decoder
  Programming 41
Terminal Count
  Trigger Sequencer 100
Test Design
  Example 128
Testing Level 131
Testing Time
  Optimizing 172
  Requester-Initiator Behavior Permutation 143
  Requester-Initiator Block Permutation 139
Trace Memory
  Example 106
  Filling 104
  Programming 104
  Programming Steps 105
Transaction Scheduler 68
Transfer Direction
  Block Permutation Properties 134
Transition Condition 97
Trigger
  Condition 98
  Counter 104
Trigger I/O
  Configuration 200
  Programming 199
Trigger Sequencer
  Components 96
  Example 100
  Feedback Counter 99
  Pattern Terms 100
  Programming 96
  Programming Steps 99
  Terminal Count 100
Tuple 123
Tuples
  Block Permutation Properties 136

U

Uncovered Permutations 173
Unoccupied Prime Number 125
USB Port
  Initialization 17

V

Value List 123
Values 123
Variation Parameters 142
  Completer-Initiator Behavior 150
  Completer-Target Behavior 147
  Requester-Target Behavior 153

W

Writing a C Program
Programming Steps 127

Publication Number: 5988-6092EN

Agilent Technologies