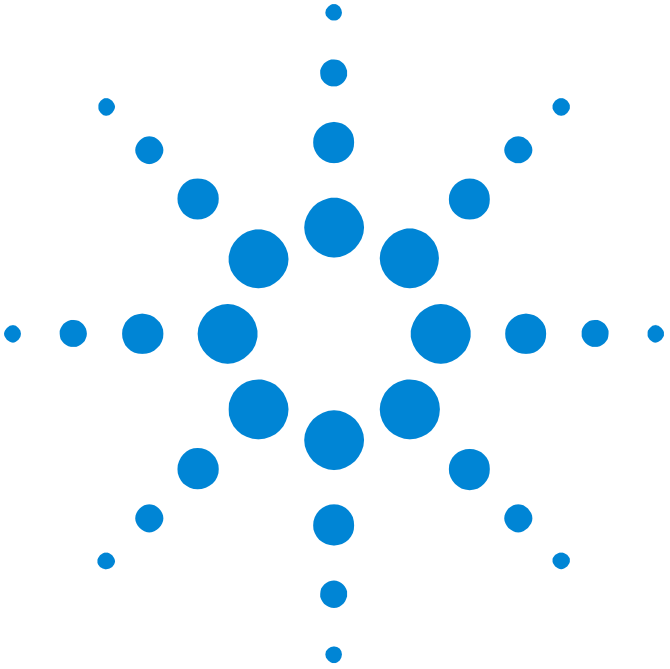


Agilent E2925B Opt. 320 C-API/PPR

Programmer's Guide

Agilent Technologies

Important Notice

Author: Anja Schauer, t3 medien GmbH

Index

Using the PPR 177

About This Guide

Programming Interface

The Agilent E2925B testcard is used for testing PCI 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 Permutator and Randomizer software (PPR)

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

Programmer's 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 E2925B testcard.

The programmer's guide contains the following chapters:

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

  It also provides two examples, one for using the C-API and one for using additional PPR functions.

- *"Programming the Framework" on page 27* provides information about the first steps to be performed in any C program, such as the testcard's connection to a control PC and its initialization.

- *"Programming the Analyzer" on page 45* provides information about programming models for all tasks of PCI analysis to monitor the PCI bus, to detect specific events, to measure and to evaluate the occurrences of signals on the bus.

- *"Programming the Exerciser" on page 75* provides information about the programming models for programming the testcard as a master and as a target device and for resources shared by both, such as data memory and compare unit.

- *"Programming the Interfaces" on page 151* provides information about the programming models for the available application interfaces, such as CPU port, static I/O port, trigger I/O sequencer, LED display and mailbox.
- *"Using the PPR" on page 177* provides an overview of the features of the software, and shows how a test program is designed and implemented.

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 E2925B PCI Exerciser and Analyzer.

The following documents are available:

User's Guides

- **Agilent E2925B Opt. 300 PCI Exerciser User's Guide**

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

- **Agilent E2925B PCI Analyzer User's Guide**

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

- **Agilent E2925B Opt. 200 PCI Performance Optimizer User's Guide**

  Provides information on how to evaluate and optimize any device under test in terms of the performance. It shows how performance measures as efficiency, data throughput, or bus utilization, allow you to compare and communicate the test results.

- **Agilent E2925B 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 E2925B Windows and Dialog Boxes Reference**

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

- **Agilent E2925B Opt. 320 C-API/PPR Reference**

  Describes all C functions, types and definitions of the application programming interface and the PPR software of the Agilent E2925B PCI testcard.

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

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 Randomizer Functionality" on page 15*.
- Error handling macros, which are needed to return error codes of C functions, are explained in *"Error Checking" on page 20*.
- Two example C programs show you how to use the C-API and the PPR software. See *"Example: Using the C-API" on page 22* and *"Example: Using the PPR" on page 23*.

Programming Interfaces

The testcard can be programmed in the following ways:

- By writing C programs

  The testcard is shipped with an application programming interface 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 *Agilent E2925B Opt. 320 C-API/PPR Reference*, together with their corresponding C function.

  For more information, refer to "*Using the Command Line Interface*" in the *Agilent E2925B Opt. 300 PCI Exerciser User's Guide*.

Hints for programming on 64 bit systems

If you plan to run the PCI 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 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).

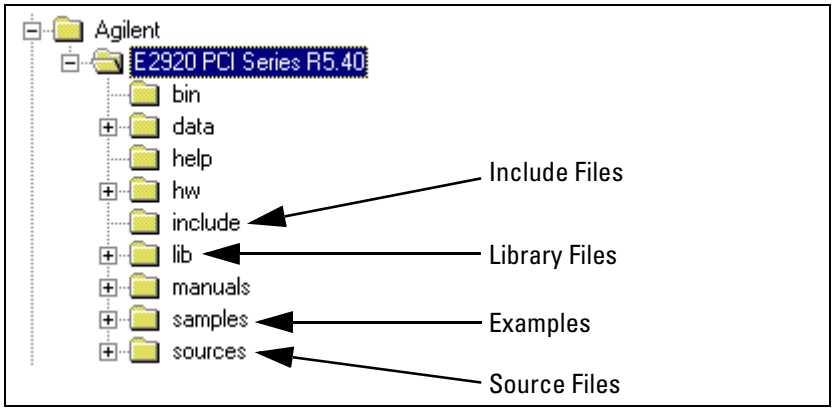
C Programming Libraries

During the installation wizard on the CD-ROM the library files, user documentation and examples to the acting control PC in your test application are installed. 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 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 Series <Revision>`.



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

- Include the referring header files into your program.
- Enter the paths to include files, library files, and/or source files into the directory settings of your 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++ 5.0.

Communication with E2920 series PCI testcards uses the E2920 series C-API. The C-API contains the necessary drivers for testcard communication. PCI C-API is available in binary form for a number of operating systems (including Windows NT), and as compilable source code for other systems. The platform and operating system determine which drivers are necessary for internal communication with the testcard and for the memory.

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 E2925B Opt. 300 PCI Exerciser User's Guide*
- *Agilent E2925B PCI Analyzer User's Guide*
- *Agilent E2925B Opt. 200 PCI Performance Optimizer User's Guide*

Protocol Permutation and Randomizer Functionality

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

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 Permutator and Randomizer software for each of them.

Computer system development requires the following steps:

- **Device bring-up and debug**

  The development process starts with the bring-up and debug 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 signal integrity tests and finishes with a **functional test phase** at the PCI protocol level.

NOTE Corner cases are exhaustive, complicated, and/or uncommon usage of PCI 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 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 interfaces of devices and motherboard to PCI traffic.

  The test checks whether the PCI 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 compatibility and again the PCI 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 Permutator and Randomizer software provides functional tests for systems and devices at the PCI protocol level and system assurance tests.

When testing devices, mainly memory controlling mechanisms can be tested by focusing on host bridges and PCI-to-PCI-bridges.

Contributions of the PCI PPR Software

In this section, the particular role of the PCI Permutator and Randomizer software is explained. Therefore, it is shown how data transfer is controlled by the Exerciser and Analyzer.

The test cases require systematically varying transfer parameters (commands, waits, burstlengths, byte enables, alignments). 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 **master block transfer memory** holds control information on how blocks are to be transferred when the Exerciser and Analyzer is used as master device (start address alignment, block size, byte enables, bus command). It also holds an entry pointing to a page in the master attribute memory, which is worked through during block transfer.
- The **master attribute memory** holds control information on master attributes on each phase of block transfer (burstlength, stepmode, wait inserting, parity/system error).
- The **target attribute memory** is used when the testcard is used as target device and holds control information on target attributes on each phase of a block transfer (parity/system error and terminations).

The PCI Permutator and Randomizer software programs these memories.

NOTE For more information on the memories, refer to *"Programming the Exerciser" on page 75*.

Operation Principles

Only the permutation constraints of attributes and block page 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 textual **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 Permutator and Randomizer software ensures that the device under test is exposed to all defined protocol variations, thus, PCI Permutator and Randomizer software determines the course of the test.

The calculation can be repeated with varying parameters, until the results of the calculation of the PCI Permutator and Randomizer software meet your testing requirements. Then the PCI Permutator and Randomizer software can **build** and **download the pages** to the Exerciser and Analyzer.

To run the test, the PCI Permutator and Randomizer software is not required. This is done by the Exerciser and Analyzer's **exerciser 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**.

Benefits

When setting up tests, you can take advantage of the following features of the Exerciser and Analyzer and the PCI 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 other.

  Tests can be set up that add as many Exerciser and Analyzers as required and letting them transfer data blocks repeatedly to generate enough traffic to stress the PCI 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 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 misbehavior on protocol and signal level using advanced listers (waveform lister, bus activity 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 from multiple PCI slots on multiple buses**

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

- **Deterministic and reproducible tests**

  In contrast to PCI 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 an error. Alternatively, the test can be continued after that error.

- **PCI protocol attribute permutations within programmable constraints**

  The software makes it possible to specify the values to be varied for each master and target attribute separately. Thus, testing time can be reduced by focusing on cases of interest. Simple problems can soon be found.

- **Detailed report**

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

- **Predictable testing time**

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

Error Checking

Each C-API function returns an error code. The error code is `0` (`B_E_OK`) if no error has occurred, otherwise it should be evaluated for error handling. Errors can be handled either by handle-based or non-handle-based error checking.

Handle-Based Error Checking

Handle-based error checking provides better error messages than non-handle-based error reporting, so it should be used whenever possible. The following macro can be simplified if your program only uses a single handle.

Note that these macro definitions rely on the use of a global variable definition `b_errtype err`.

Used Macro

Use a macro similar to this for handle-based error checking:

```
#define C1(handle, x) if ((err = x) != B_E_OK)\
{
  printf ("%s (line %d)\n", \
  Best Last ErrorStringGet(handle), __LINE__);\
  return -1;
}
```

This macro can be called in either one of the following ways:

- `C1(handle_1,BestMasterGenPropDefaultSet( handle ));`
- `err=BestMasterGenPropDefaultSet( handle ); C1(handle_1, err);`

Simplified Version of Handle-Based Error Checking

In case you are using a single handle identified by the name *handle*, use the following macro:

```
#define C1(x) if ((err = x) != B_E_OK) \
{
  printf (”%s (line %d)\n”, \´
  Best Last ErrorStringGet(handle), __LINE__);\
  return -1;
}
```

This macro can be called in either one of the following ways:

- `C1(BestMasterGenPropDefaultSet( handle ));`
- `err=BestMasterGenPropDefaultSet( handle ); C1(err);`

Non-Handle-Based Error Checking

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

- `BestDevIdentifierGet()`
- `BestPCICfgMailboxReceiveRegRead()`
- `BestPCICfgMailboxSendRegWrite()`

Handle Initialization

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

- `BestOpen()`

Used Macro

Use all the functions with a macro similar to this:

```
#define C(x) if ((err = x) != B_E_OK) \
{
  printf ("%s (line %d)\n", \
  BestErrorStringGet(handle), __LINE__);\
  return -1;
}
```

This macro can be called in either one of the following ways:

- `C(BestDevIdentifierGet(ven, dev, no, &devid));`
- `err=BestDevIdentifierGet(ven, dev, no, &devid); C(err);`

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

Example: Using the C-API

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

```
#include <stdio.h>
#include <mini_api.h>


define CH(x) if ((err = x)) != B_E_OK\
{printf ("%s (line %d)\n", \
BestLastErrorStringGet(handle), __LINE__);\
return -1; }


#define C(x) if ((err = x)) != B_E_OK\
{printf ("%s (line %d)\n", \
BestErrorStringGet(handle), __LINE__);\
return -1;


int main ( )

{
   b_errtype err;
   b_charptrtype version_string;
   b_handletype handle;

   /* Open the communication session to testcard, initialize  */
   /* internal structures. */
   err = BestOpen(&handle,B_PORT_RS232,B_PORT_COM1); C(err);

   /* If using RS232, set baud rate: */
   err = BestRS232BaudRateSet(handle,B_BD_57600); CH(err);

   /* For example:*/
   /* Read product & serial number from testcard. */
   err = BestVersionGet (handle, B_VER_PRODUCT, &version_string);
   CH(err);
   printf("Product: %s\n", version_string);  err = BestVersionGet
   (handle, B_VER_SERIAL, &version_string); CH(err);
   printf("Serial#: %s\n", version_string);

   /* Close the session to deallocate memory. */
   err = BestClose(handle); CH(err);
}
```

Example: Using the PPR

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

WARNING This program fragment writes data to system memory. To run this program in a real environment, a line that allocates the required memory must be added.

```
#include <stdio.h>
#include <mini_api.h>
#include <ppr.h>

#define CHECK { if (status != B_E_OK)    \
         {printf ("ERROR line %d, %s\n", __LINE__,
          BestErrorStringGet(status) ); return -1;}\
         }
#define WARN { if (status != B_E_OK)    \
         {printf ("WARNING
         line %d, %s\n", __LINE__, BestErrorStringGet(err) );} \
         }


int main ( void )
{
 b_errtype       status;
 b_handletype    handle;
 b_int32         status_reg;
 b_int32         errbit;
 b_int32         count;
 b_charptrtype   errtext;
 b_int32         blockruns;

/* Open the communication session to testcard on Fasthost
interface.*/
   status = BestOpen( &handle, B_PORT_PARALLEL,
   B_PORT_LPT1); CHECK;

/* Set attribute mode to sequential. */
   status=BestMasterGenPropSet( handle,
                                B_MGEN_ATTRMODE,
                                B_ATTRMODE_SEQUENTIAL); CHECK;

/* Initialize PPR. */
   status=BestPprInit (handle); CHECK;
```

```
/* Set up generic PPR properties. */
   status=BestPprGenPropSet( handle, BPPR_GEN_BUSWIDTH, 32 );
CHECK;

/* Block permutation.
 * Data is transfered from testcard internal address 0
 * to busaddress 0xb8000 (video memory). */

 printf ("programming Block permutation\n");

 status=BestPprBlockPermPropSet( handle,
                 BPPR_BLK_DIR,
                 BPPR_DIR_WRITE ); CHECK;

 status=BestPprBlockPermPropSet( handle,
                 BPPR_BLK_BUSADDR,
                 0x0b8000 ); CHECK;

 status=BestPprBlockPermPropSet( handle, BPPR_BLK_INTADDR, 0 );
CHECK;
 status=BestPprBlockPermPropSet( handle, BPPR_BLK_NOFDWORDS, 64 );
CHECK;
 status=BestPprBlockPermPropSet( handle, BPPR_BLK_ATTRPAGE, 2 );
CHECK;
 status=BestPprBlockPermPropSet( handle, BPPR_BLK_PAGENUM, 1 );
CHECK;
 status=BestPprBlockPermPropSet( handle, BPPR_BLK_PAGESIZEMAX, 60);
CHECK;
 status=BestPprBlockPermPropSet( handle, BPPR_BLK_CACHELINE, 4);
CHECK;

/* Block variation properties. */
 status=BestPprBlockVariationSet( handle,
                  BPPR_BLK_ALIGN,
                  "(%16=0), (%16=4), (%16=8), (%16=12), (%32=0)",
                  BPPR_ALG_PERM); CHECK;

 status=BestPprBlockVariationSet( handle,
                  BPPR_BLK_SIZE,
                  "4,8,16",
                  BPPR_ALG_PERM ); CHECK;
 status=BestPprBlockVariationSet( handle,
                  BPPR_BLK_CMDS,
                  "mem_write, mem_writeinvalidate",
                  BPPR_ALG_PERM ); CHECK;
 status=BestPprBlockGenerate( handle ); CHECK;
```

```
/* Master attribute permutations. */
 printf ("Programming master attr. permutation\n");
 status=BestPprMAttrPermPropSet( handle,
                  BPPR_MA_PAGENUM, 2 ); CHECK;

 status=BestPprMAttrPermPropSet( handle,
                  BPPR_MA_PAGESIZEMAX, 49 ); CHECK;
 status=BestPprMAttrVariationSet( handle,
                  B_M_LAST,
                  "4, 8, 32",
                  BPPR_ALG_PERM ); CHECK;

 status=BestPprMAttrVariationSet( handle,
                  B_M_WAITS,
                  "0, 1, 3, 8",
                  BPPR_ALG_PERM ); CHECK;

 status=BestPprMAttrVariationSet( handle,
                  B_M_STEPS,
                  "0, 7",
                  BPPR_ALG_PERM ); CHECK;

 status=BestPprMAttrVariationSet( handle,
                  B_M_TRYBACK,
                  "true, false",
                  BPPR_ALG_PERM ); CHECK;

/* Generate master attributes page. */
 status=BestPprMAttrGenerate( handle ); CHECK;

/* Print a report w/o target attributes. */
 status=BestPprReportPropSet (handle, BPPR_REP_TA, 0); CHECK;
 status=BestPprReportPropSet (handle, BPPR_REP_TACONTENT, 0);
CHECK;
 status=BestPprReportFile(handle, "report.txt"); CHECK;
```

```
/* Obtain number of blockruns necessary for complete coverage. */
 status=BestPprMAttrResultGet( handle,
                  BPPR_MA_RUNS,
                  &blockruns );

 printf ("Running master %u times\n", blockruns);
 for (count=0; count<blockruns; count++)
 {
  status=BestMasterBlockPageRun(handle, 1); CHECK;

  do
  {
   status=BestStatusRegGet(handle, &status_reg); CHECK;
  }
  while ( (status_reg & 0x01) );
  if (status_reg & 0x80)
  {
   printf ("Test failed, master abort has occured!\n");
   break;
   }
 }

/* Get protocol errors. */
 if (status_reg & 0x10)
 /* protocol error occured */
 {
  status=BestObsStatusGet (handle, B_OBS_ACCUERR, &status_reg);
CHECK;

  printf("The following protocol errors habe been detected:\n");
  for (errbit=1;
   errbit<=0x010000000; errbit >>=1)
  {
   if (status_reg & errbit)
   {
   status=BestObsErrStringGet (handle, errbit, &errtext); CHECK;
   printf ("%s\n", errtext);
   }
  }
 }

 /* Close the session, deallocate memory. */
 status=BestPprDelete( handle ); CHECK;
 status=BestClose( handle ); CHECK;

}
```

Programming the Framework

The following sections provide information about the testcard's connection to a control PC and its initialization. These are the first steps to be performed in any C program for the testcard.

- *"Connection and Initialization" on page 28* shows you to set up and specify the control interface(s) and how to establish the connection.
- *"Administration" on page 33* gives information about performing several checks, such as checks for enabled capabilities of the testcard, for current versions of testcard's components or for system information.

  Here you get also information about resource locking.
- *"Power-Up and Reset Control" on page 37* 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 your system under test. It can also help when the testcard hangs and you need to unlock it.
- *"Card Status Register Access" on page 42* gives information about using the testcard's status register.

  This information is useful for evaluating test results or for debugging and evaluating errors.

Connection and Initialization

When executing a C program for the testcard, the testcard and the connections must be initialized. The testcard can be controlled via PCI port, RS-232 serial interface or Fast Host Interface. Some typical initialization routines for each type of control connection are shown in *"Examples" on page 30*.

PCI Port

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

No system resources are required to program the testcard. This is especially useful when the testcard is used as a passive observer (protocol checker or performance monitor) and is not authorized to change the system configuration of the system under test (for example, memory mapping).

RS-232 Serial Interface

The RS-232 serial interface provides an easy-to-use control interface, which is available on all PCs and notebook computers. It can be run at 2400, 4800, 9600, 19200, 38400, and 57600 baud (8 bit data, 1 stop bit, no parity).

Fast Host Interface Port

The Fast Host Interface port provides an easy-to-install connection to a standard PC with higher throughput than an RS-232 interface in both read and write directions.

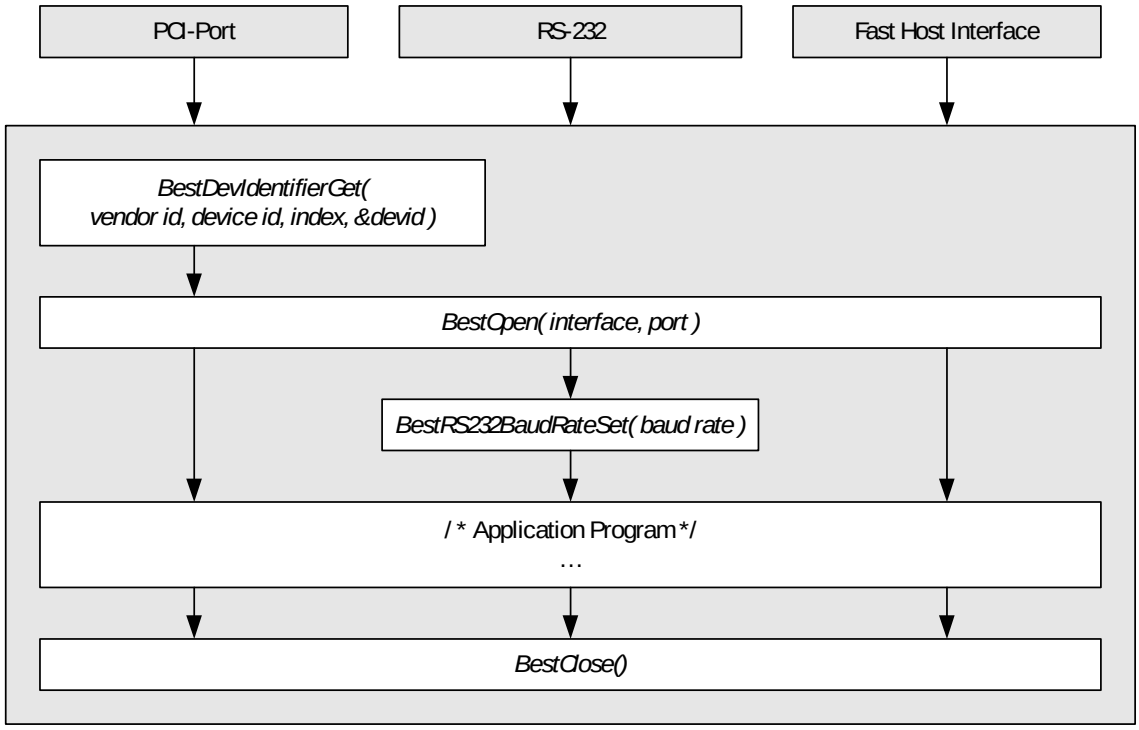
The control PC must be equipped with the Fast Host Interface card coming with the PCI Analyzer and connected to the parallel port on the testcard.

Specification

Maximum transfer rate: 4 MB/s (using the Fast Host Interface).

Functions Overview

The following figure shows the available functions used for connecting and initializing the testcard. This figure also shows the integration of these functions into the test program.



Programming Steps

Initializing the Exerciser and Analyzer testcard requires the following steps:

1. If the **PCI Bus** is used as the controlling interface port, use *BestDevIdentifierGet* to get the device number of the testcard.

   This device number is used in *BestOpen* for device identification.

2. Initialize internal structures and variables for the control port and establish the connection. Use *BestOpen*.

3. If the **RS-232** serial interface is used, set the baud rate with *BestRS232BaudRateSet*.

4. Insert your application code.

   Parts of your program may communicate with different resources via different ports on the testcard. Therefore, resources must be locked while they are used and unlocked after they have been used. Use *BestResourceLock* and *BestResourceUnlock*.

5. Close the connection and deallocate the session memory with *BestClose*.

Examples

The following examples show the programming steps required to initialize the testcard, and to set up a control connection to it. An example is given for each type of control connection:

- serial
- parallel
- PCI port

Note that for clearness and convenience the errors are handled assuming that only one connection has been opened and the session handle is named “handle”. This also enables compatibility with previous program versions.

Serial Port Example

Task In this example, a connection to the testcard is opened using the serial port and the baud rate is set to 57600 bps.

Implementation

```
#include <stdio.h>
#include <mini_api.h>

int main ( )
{
b_errtype err;
b_handletype handle;

/*Initialize port internal structs and variables*/
err=BestOpen(&handle,B_PORT_RS232,B_PORT_COM1); C(err);

/*Set baud rate to 57600*/
err=BestRS232BaudRateSet(handle,B_BD_57600); C(err);

/* Start of application program code, for example, locking the
exerciser */
err=BestResourceLock (handle,B_RESLOCK_EXERCISER); C(err);

/* This line represents the application program for the exerciser
*/

/* After exerciser application program's end, unlock the exerciser
*/
err=BestResourceUnlock(handle, B_RESLOCK_EXERCISER); C(err);

/* close the session and deallocate memory*/

err=BestClose(handle); C(err);

}
```

Fast Host Interface Example

Task The following example shows how to open a connection to the testcard using the Fast Host Interface.

Implementation

```
#include <stdio.h>
#include <mini_api.h>

int main ( )
{
b_errtype err;
b_handletype handle;

/*Initialize port internal structs and variables*/
err=BestOpen(&handle,B_PORT_FASTHIF,0); C(err);

/* Start of application program code, for example, locking the
exerciser */
err=BestResourceLock (handle,B_RESLOCK_EXERCISER); C(err);

/* This line represents the application program for the exerciser
*/

/* After exerciser application program's end, unlock the exerciser
*/
err=BestResourceUnlock(handle, B_RESLOCK_EXERCISER); C(err);

/* close the session and deallocate memory*/
err=BestClose(handle); C(err);
}
```

PCI Bus Example With Two Testcards and Reading Out Capabilities

Task

The following example opens a connection to two testcards using the PCI interface. *BestDevIdentifierGet* is used to request the **device identifier** of each testcard. This device identifier is then used to open the connection to the respective testcard.

The third parameter of *BestDevIdentifierGet* is an **index** used for testcard identification when multiple testcards are used.

The example also shows how to read out the testcard's capabilities.

Implementation

```
#include <stdio.h>
#include <mini_api.h>

int main ( )
{
b_errtype err;
b_handletype handle1, handle2;
b_int32 devid;
b_int32 capability_code;

/*Get device number devid of first testcard
The index (number=0) can be used to distinguish between
multiple testcards*/
err=BestDevIdentifierGet(0x103C, 0x2926, 0, &devid); C(err);

/*Initialize port internal structs and variables*/
err=BestOpen(&handle1, B_PORT_PCI_CONF, devid); C(err);

/*Repeat for the second testcard (number=1)*/
err=BestDevIdentifierGet(0x103C, 0x2926, 1, &devid); C(err);
err=BestOpen(&handle2, B_PORT_PCI_CONF, devid); C(err);

/* Application program code, check here for capabilities */
err=BestCapabilityRead(handle1, &capability_code); C(err);
if
(capability_code & (B_CAPABILITY_EXERCISER | B_CAPABILITY_ANALYZER))
      {
      printf("testcard1:");
      printf("exerciser capability enabled or ");
      printf("analyzer capability enabled (or both) !\n");
      }
else
      {
      printf("testcard1: Neither exerciser nor analyzer capability
      enabled !\n");
      }
```

```
err=BestCapabilityRead(handle2, &capability_code); C(err);

if
(capability_code & ( B_CAPABILITY_EXERCISER | B_CAPABILITY_ANALYZER )
      {
      printf("testcard2:");
      printf("exerciser capability enabled or ");
      printf("analyzer capability enabled (or both) !\n");
      }
else
      {
      printf("testcard2: Neither exerciser nor analyzer capability
      enabled !\n");
      }

/* close the session and deallocate memory*/
BestClose(handle1); C(err);
BestClose(handle2); C(err);

}
```

Administration

You can prepare the following during the initialization phase of your C program.

Version Checking

Before your program starts actions on the testcard, you can let it check for versions of the testcard's components.

To ensure compatibility of hardware, firmware and C-API software, you can check for the versions of the following components of the testcard.

- Card's product number
- Hardware serial number
- Card version
- Core BIOS
- Firmware version and date
- XILINX FPGA (Field Programmable Gate Array) chain architecture
- C-API version

Resource Locking

The C program can access different resources on the testcard via different interfaces. To guarantee proper operation, you can lock different resources to different interfaces. This prevents the resources from being simultaneously accessed via different ports.

Capability Checking

In offline mode, the C program can run without an testcard and/or without the required product capabilities. This may be useful, for example, for testing or demo purposes. You can check for available product capabilities in your program's initialization phase.

The C-API allows you to check whether or not the following capabilities are available:

- All capabilities
- No capabilities
- Analyzer
- Exerciser
- Host interface (CPU port, static I/O, host access functions)
- 64-bit PCI
- 66-MHz PCI (for exerciser and analyzer)
- Trace memory sizes
- Performance measures

It is a good idea to have this information available before you call support.

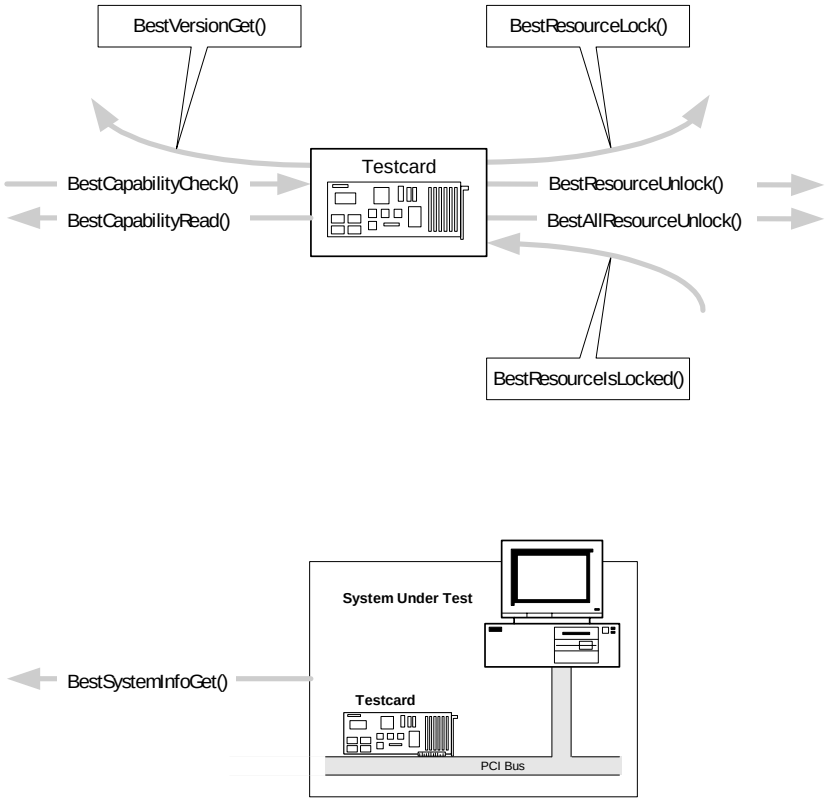
If your hardware does not have one or more of the above capabilities, you can still develop or test in offline mode, because this mode does not use a physical port and, therefore, does not require hardware.

System Checking

The C-API also enables you to request the buswidth (32 or 64 bits) and speed (33 or 66 MHz) of the PCI system under test.

Functions Overview

The following figure shows all functions available for administration purposes regarding to the testcard and the system under test.



Programming Options

The administration functions of the testcard allow the following options:

- To ensure compatibility of hardware, firmware and C-API software, or in case of support calling, check the version of testcard's components.

  Use *BestVersionGet*.

- To lock different resources to different interfaces, use *BestResourceLock*.
- To unlock resources or to check whether resources are locked, use *BestResourceUnlock*, *BestAllResourceUnlock* and *BestResourceIsLocked*.
- To check for a special available capability such as Analyzer or Exerciser, use *BestCapabiltityCheck*.

  To get information about all available capabilities, use *BestCapabilityRead*.

- To get system information, such as buswidth and busspeed, use *BestSystemInfoGet*.

Examples

Task

The following code fragments give examples for administration purposes.

Version Checking

```
/* Read product & serial number from testcard */
err=BestVersionGet (handle, B_VER_PRODUCT, &version_string); C(err);
printf("Product: %s\n", version_string);

err=BestVersionGet (handle, B_VER_SERIAL, &version_string); C(err);
printf("Serial#: %s\n", version_string);
```

Resource Locking

```
/* Locking the exerciser */
err=BestResourceLock (handle,B_RESLOCK_EXERCISER); C(err);

/* This line represents the application program for the exerciser */

/* After exerciser application program's end, unlock the exerciser */
err=BestResourceUnlock(handle, B_RESLOCK_EXERCISER); C(err);
```

Capability Checking

```
/* Application program code, check here for capabilities*/
err=BestCapabilityRead(handle1, &capability_code); C(err);
if
(capability_code & (B_CAPABILITY_EXERCISER | B_CAPABILITY_ANALYZER))
      {
      printf("testcard1:");
      printf("exerciser capability enabled or ");
      printf("analyzer capability enabled (or both) !\n");
      }
else
      {
      printf("testcard1: Neither exerciser nor analyzer capability
      enabled !\n");
      }

err=BestCapabilityRead(handle2, &capability_code); C(err);
if
(capability_code & (B_CAPABILITY_EXERCISER | B_CAPABILITY_ANALYZER))
      {
      printf("testcard2:");
      printf("exerciser capability enabled or ");
      printf("analyzer capability enabled (or both) !\n");
      }
else
      {
      printf("testcard2: Neither exerciser nor analyzer capability
      enabled !\n");
      }
```

System Checking

```
/* Checking for the bus speed */
berr=BestSystemInfoGet(handle,B_SINFO_BUSSPEED,&BusSpeed); C(err);
```

Power-Up and Reset Control

The behavior of the testcard during power-up or reset can be controlled by programming power-up properties. Controlling this behavior is then needed when writing C programs that focus on the power-up behavior of the system under test, or when the PCI bus hangs making a reset necessary.

The following properties are available to control power-up and reset:

- *Power-Up*

  After power-up, the testcard is completely reset. The testcard's configuration space header settings control the behavior of the testcard after power-up with property B_PU_CONFRESTORE. Set this property according to your test environment before you power down the testcard:

  – If the testcard is used in a PCI system **with** BIOS, then the BIOS-programmable bits in the base address registers of the testcard should be set to 0 for power-up—that is: these bits should not be restored from the settings before power-up. The BIOS can reprogram them when allocating memory resources during system configuration.

  – In a system **without** BIOS, you must set these bits to allocate memory resources. They should be programmed in such a way (using the power-up properties) that they do not need to be reprogrammed after each power-up.

  For information on the configuration space header, refer to "Configuration Space Header" in the *PCI Exerciser User's Guide*.

- *PCI Reset*

  Normally, a PCI reset is issued by the system controller of the system under test and has the same effect as power-up.

  However, property B_BOARD_RSTMODE can be used to prevent the testcard from being completely reset, for example, to avoid loss of data or change of states. That means that only the internal state machines and the target are reset and initialized; the master and trace memory are not automatically reset because they are controlled by their own power-up properties.

- *Board Reset* and *Statemachine Reset* commands

  These commands issue an testcard reset from the C program:

  The *Board Reset* command is used to establish a defined state of the testcard before the C program actually begins. The command has the same effect as power-up, however, it does not affect the configuration space settings or internal state machines.

  The *Statemachine Reset* command can be used before recovering data from the testcard if the testcard does not react anymore because of a hanging PCI bus.

Restoring Settings after Resets

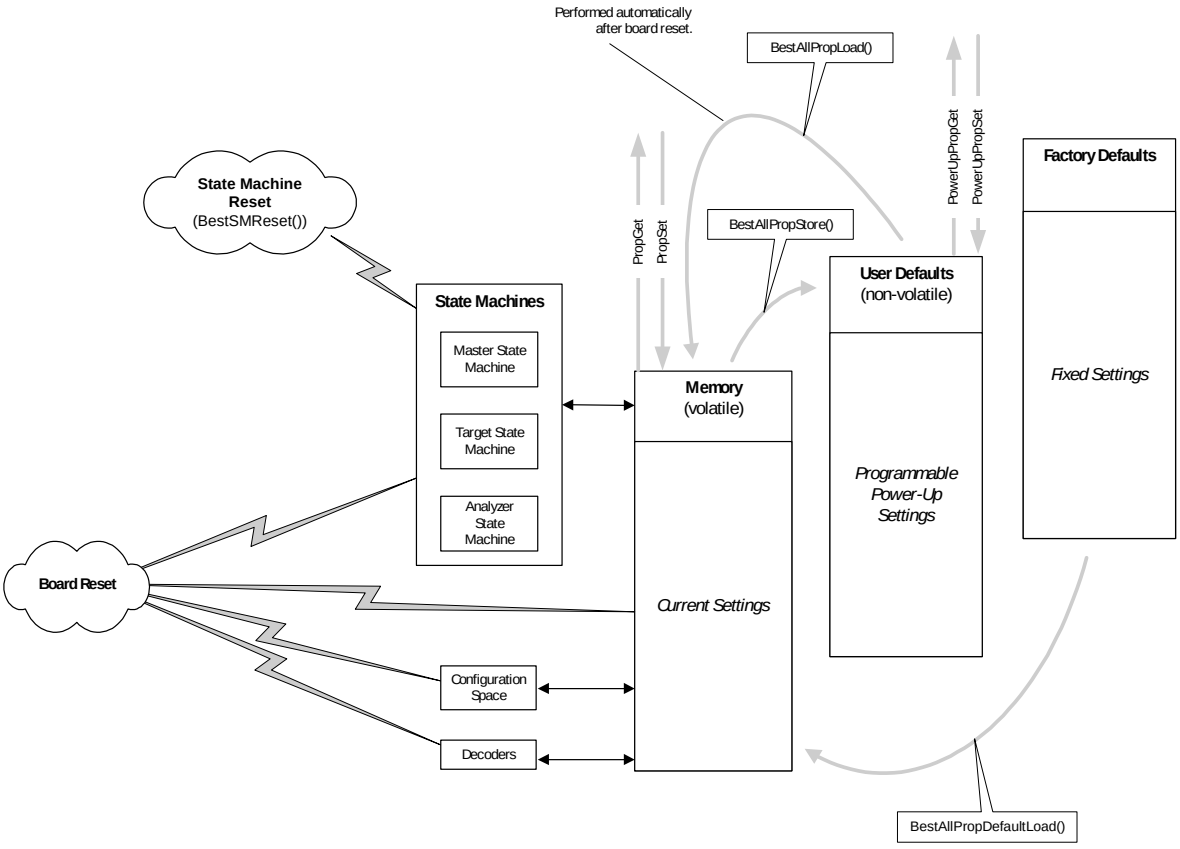
The testcard's behavior is determined by property settings in its **memory**. These settings are currently influenced internally by programming, or externally by the test flow.

Because the testcard's memory is volatile, the current property settings are lost after the testcard is reset. To ensure a deterministic behavior during power-up, power-up settings are automatically loaded from a non-volatile, programmable memory. These settings are referred to as **user defaults** and allow a programmable power-up behavior of the testcard.

However, the user defaults could also be set in a way that causes start-up problems with the system under test or the testcard. In this case, **factory defaults** can be used instead.

Functions Overview

The following figure shows all functions used to program power-up and reset behavior of the testcard and displays all memories controlled by these functions.



Programming Options

The power-up control of the Exerciser and Analyzer testcard allows the following options:

- To program the user defaults, store the current settings as power-up defaults with *BestAllPropStore*.
- To use the user defaults as current settings, load them to the memory with *BestAllPropLoad*.
- To use the factory defaults as current settings, load them to the memory with *BestAllPropDefaultLoad*.
- To issue a board reset and a state machine reset, use *BestBoardReset* and *BestSMReset*.
- To determine whether a PCI reset causes a board reset or a state machine reset, set the “board mode” with *BestBoardPropSet*.

Examples

The testcard's behavior is determined by property settings in its **memory**. These settings are currently influenced internally by programming, or externally by the test flow.

The following examples shows how to define the power-up behavior of the testcard:

- *"Factory Defaults for Power-Up" on page 40*

  The user defaults can be set in a way that causes start-up problems with the system under test or the testcard. In this case, **factory defaults** can be used instead.

- *"User Defaults for Power-Up" on page 41*

  Because the testcard's memory is volatile, the current property settings are lost after the testcard is reset. To ensure a deterministic behavior during power-up, power-up settings are automatically loaded from a non-volatile, programmable memory. These settings are referred to as **user defaults** and allow a programmable power-up behavior of the testcard.

Factory Defaults for Power-Up

Task The following example shows how to program the testcard to use the factory defaults for power-up.

Implementation

```
/* Load the factory defaults as current settings. */
err=BestAllPropDefaultLoad( handle ); C(err);

/* Load the current settings (now acting as the factory defaults)
as user defaults. */
err=BestAllPropStore( handle );C(err);

/* Reset the board and uses the factory defaults as power-up
settings. */
err=BestBoardReset( handle );C(err);
```

User Defaults for Power-Up

Task Instead of using the factory defaults, you can program *any* user defaults according to your specific test requirements. All functions described in the C-API reference the names of which end with `...PropSet()` write current settings to the memory. This is shown in the following example.

Implementation

```
int main (int argc, char *argv[])
{
   b_errtype err;
   b_handletype handle;

   err=BestOpen(&handle,B_PORT_PARALLEL,B_PORT_LPT2); C(err);
   err=BestConnect ( handle ); C(err);

   /* Set vendor and device id:*/
   err=BestConfRegSet(handle, 0x00, 0x2925103c); C(err);

   /* Make Device and Vendor ID read-only.*/
   err=BestConfRegMaskSet(handle, 0x00, 0x00000000); C(err);

   /* Read/write bits will have their factory default values at
      powerup. */
   err=BestPowerUpPropSet(handle, B_PU_CONFRESTORE, 0); C(err);
   err=BestAllPropStore(handle); C(err);

   /* Disconnect from the current port.*/
   err=BestDisconnect (handle); C(err);

   /* Close the session and deallocate memory.*/
   err=BestClose(handle); C(err);
   return 0;
}
```

Card Status Register Access

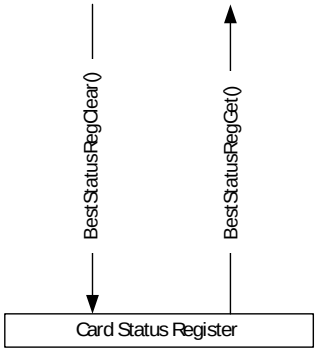
The testcard status register can be used, for example, to evaluate the test result after the test run, and to debug and evaluate errors. The bits show:

- the exerciser status: master run, active target, data compare error, or master block abort.
- the analyzer status: protocol error, trace memory is running (recording), asserted interrupts.
- whether a C function returned an error (error code not equal to zero).
- whether a high level test function (see Built-In Test Functions) has failed.
- whether another on-board application has failed.
- whether a PCI interrupt has occurred.

For the content of the status register, refer to "*Testcard Status Register*" in the *Agilent E2925B Opt.320 C-API/PPR Reference*.

Functions Overview

The following figure shows the functions available to access the status register of the testcard.



Programming Steps

Executing a test program requires access to the testcard status register as follows:

1. Before executing the test program, clear all bits of the testcard status register with *BestStatusRegClear* to ensure a definite register condition. All bits are set to 1.
2. After executing the test program, read the whole content of the testcard status register with *BestStatusRegGet*.

Example

The following lines show how to poll the status register to detect the end of a master run.

```
do
{
        err=BestStatusRegGet(handle, &statusreg); C(err);
}
while(statusreg & 0x01);
```

Programming the Analyzer

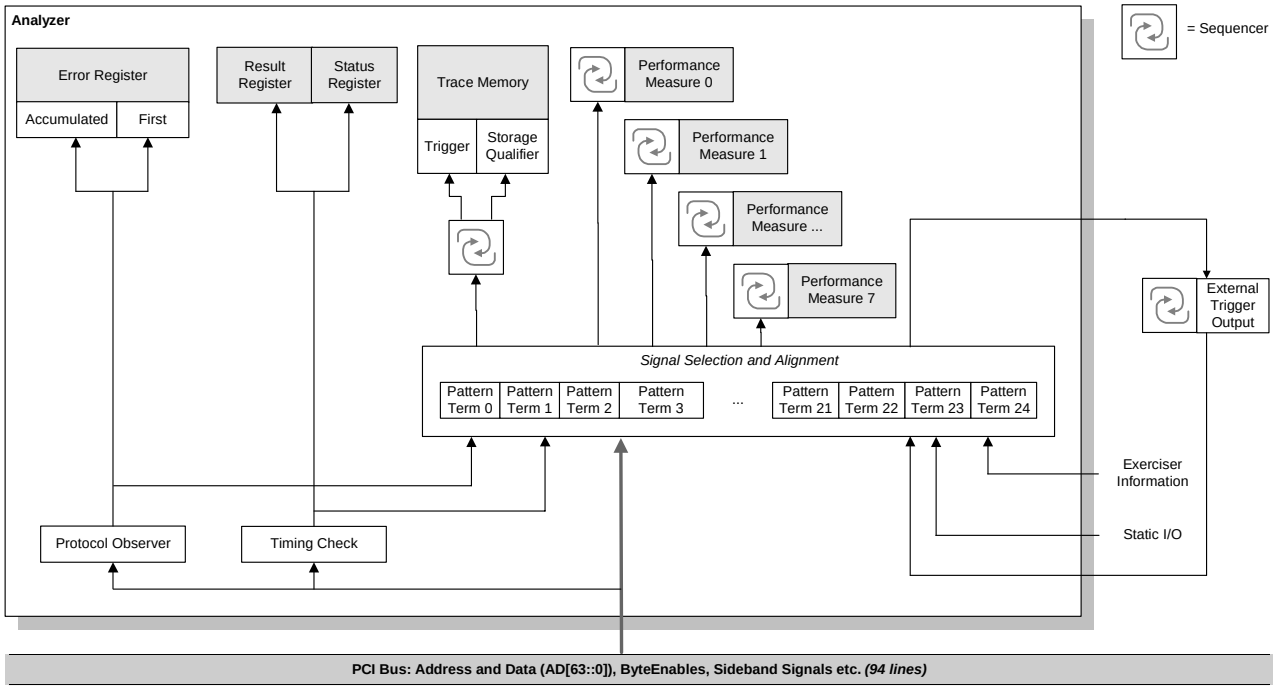
The task of the PCI analysis is to monitor the PCI bus, to detect specific events, to measure and evaluate the occurrences of signals on the bus. The following sections explain how to program the components of the testcard's analyzer fulfilling the different tasks:

- *"Protocol Observer Programming" on page 47* explains how to mask rules to be observed and how to read the observer result registers.
- *"Timing Check Programming" on page 49* explains how to set up the timing check, and how to get the results.
- *"Programming the Pattern Terms" on page 52* explains all types of pattern terms, and how to use and program them.
- *"Sequencer Programming" on page 55* 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 sequence is provided to show how to program the sequencers.
- *"Performance Measurement Programming" on page 64* explains how to program the performance measures.
- *"Trace Memory Programming" on page 70* explains how to use the trace memory and how to program its sequencer and the storage qualifier. How to upload and evaluate the contents of the trace memory is also shown.

Analyzer Components

The following figure shows the components of the Analyzer with its inputs and outputs and from where the results of the analysis can be taken.



Inputs are:

- 94 line PCI signals: address and data lines, byte enables, sideband signals and so forth
- additional information generated by the observer
- exerciser signals: master and target marker, different outputs from their statemachines, and so forth.
- 12 line external trigger input
- 8 line static I/O

Results can be taken from:

- Result and Error Registers
- Trace Memory
- Performance Measures

The **outputs** can be used to trigger external devices.

The analyzer and exerciser of one testcard can be used in parallel. This allows you to set up the analyzer to monitor **exerciser** transactions.

Protocol Observer Programming

The protocol observer monitors 53 different protocol rules simultaneously. The protocol rules refer to PCI 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:

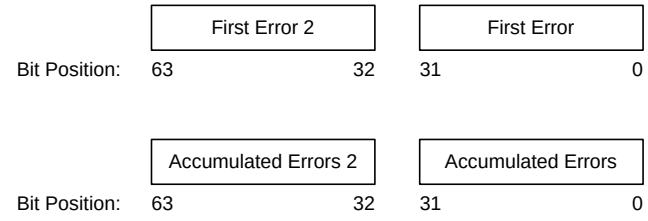
- Bits for the protocol rule violations that have occurred first. Often the first rule violations are the reason for subsequent rule violations.

  Each individual rule can be masked from being detected as “first rule violation”. This allows you to exclude rule violations prior to those of your interest 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 it into a text string describing the violated rule.

Further Use

A detected protocol violation can:

- be used as input for pattern terms (see *“Programming the Pattern Terms” on page 52*).
- trigger the trace memory (see *“Trace Memory Programming” on page 70*).

  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 data does not match the PAR signal. Using a storage qualifier allows for storing only the incorrect data phases.

Functions Overview

The Agilent E2925B testcard's programming interface provides functions for programming the protocol observer. The available functions and their usage is shown by describing the programming steps.

Programming Steps

Programming the protocol observer requires the following steps:

1. Set the observer properties.

   To ensure that all protocol rules will be observed, set all mask bits to 0.

   Use *BestObsDefaultSet*.

2. Specify a mask.

   Set the protocol rules to be ignored in the "first error register".

   Use *BestObsMaskSet*.

3. Request the protocol errors.

   To determine whether rules have been violated, check whether the "first error" result registers in the observer status register hold a value.

   Use *BestObsStatusGet*.

4. Request the error string.

   To read the errors, convert the information read from the registers into a text string and send it, for example, to a file or to standard output.

   Use *BestObsErrResultGet*.

Example

Task

Set up the protocol observer to mask the `PARITY_1` rule, read the detected protocol errors and print the error string.

Implementation

```
/* Set the observer properties to their default values. */
err=BestObsPropDefaultSet(handle); C(err);

/* Specify a Mask: Mask the rules PARITY_1 by setting their bits in
the mask register to 1 */
err=BestObsMaskSet(handle, B_R_PARITY_1, 1); C(err);

/* Check the value in the "first error" result registers in the
observer status register. */
err=BestObsStatusGet( handle, B_OBS_FIRSTERR, &firsterr1); C(err);
err=BestObsStatusGet( handle, B_OBS_FIRSTERR2, &firsterr2); C(err);
```

```
/* Read the lower bits of the accumulated error register. */
err=BestObsStatusGet(handle, B_OBS_ACCUERR, &accuerr1); C(err);

/* Read the upper bits of the accumulated error register. */
err=BestObsStatusGet(handle, B_OBS_ACCUERR2, &accuerr2); C(err);


/* Clear the observer status register. */
err=BestObsStatusClear(handle);C(err);

/* Print the error string using the error register values. */
err=BestObsErrResultGet( handle, accuerr1, accuerr2, &errtxt); C(err) ;
printf ("Protocol error: %s\n", errtxt);
```

Timing Check Programming

The testcard checks the PCI bus for setup and hold timing violations in real-time. Checking is always performed while the testcard is powered. You can disable individual signals if their observation interferes with your test.

For a list of all available signals, refer to "*b_signaltype (for Timing Check)*" in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

NOTE At present, the timing check is only available for 33 MHz PCI busses.

Functions Overview

The Agilent E2925B testcard's programming interface provides functions for programming the timing checker. The available functions and their usage is shown by describing the programming steps.

Programming Steps

Programming the timing checker requires the following steps:

1. Enable all signals and preset the set-up and hold time to the values according to the PCI Specification.

   Use *BestTimCheckDefaultSet*.

2. To set-up and hold time to values other than the PCI Specification defaults, set the generic timing check properties to allow changes.

   Use *BestTimCheckGenPropSet*.

3. To select the signals to be checked for your test, mask the signals that are not relevant.

   Use *BestTimCheckMaskSet*.

4. Set up the timing parameters in the preparation register.

   Use *BestTimCheckPropSet*.

5. Write the settings to the testcard.

   Use *BestTimCheckProg*.

6. To determine whether the PCI frequency is stable enough for a proper timing check, read the timing check status .

   Use *BestTimCheckStatusGet*.

NOTE The result registers of the timing check are cleared automatically and the check is continued with the new parameters.

7. Determine whether a timing violation has occurred and print the textual report.

   Use *BestTimCheckResultGet*.

Example

Task Program a timing check against a set-up time of 6 ns and a hold time of –250 ps at a bus speed of 33 MHz.

Implementation

```
/* Set the timing check to default values. */
BestTimCheckDefaultSet(handle);

/* Set the generic timing check property. */
BestTimCheckGenPropSet(handle,B_TCGEN_SPEC,0);

/* Set up the timing parameters in the preparation register and
write them to the card. */
BestTimCheckPropSet(handle,B_TC_SETUP_TIME,6000);
BestTimCheckPropSet(handle,B_TC_HOLD_TIME,250);
BestTimCheckPropSet(handle,B_TC_HSIGN,1);

BestTimCheckProg(handle);
```

At this point of the program, it is expected that some traffic can be found on the PCI bus to see whether signals are violated.

```
/* Read the timing check status to determine whether the PCI
frequency is stable enough for a proper timing check. */

BestTimCheckStatusGet(handle,B_TC_TCSTAT,&status);
if (status & B_TC_ERROR)
  {
    printf("Timing checker data incorrect because frequency has
    changed !\n");
    exit (1);
  }

/* Determine whether a timing violation has occurred and print a
textual report. */

if (status & B_TC_VIOLATION)
  {
    printf("Timing violation occurred\n");
    BestTimCheckResultGet(handle,&errorreport);
    printf("%s\n",errorreport);
  }
```

Programming the 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 (`pt0` ... `pt23`). For a list of valid pattern term identifiers, see "*Pattern Term Identifiers*" in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

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

Pattern `pt0` can be used either as a standard or as a transitional pattern term. Different operators are available for standard and transitional pattern terms.

Using Pattern Terms

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

- as input for sequencers, for example, the trace memory trigger sequencer (see *"Sequencer Programming" on page 55*).
- for storage qualification for the trace memory (see *"Trace Memory Programming" on page 70*).
- when counting bus events for performance analysis (see *"Performance Measurement Programming" on page 64*).
- for master conditional start based on the detection of a specific event on the PCI bus (see *"Master Run" on page 103*).

As input, the pattern terms can use all the signals specified in "*b_signaltype (List of Signals)"* in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

24 pattern terms (named `pt0` … `pt23`) are implemented on the testcard. `pt0` can be used as input for the trace memory trigger only.

Types of Pattern Terms

By default, all pattern terms are standard pattern terms. However, the pattern term `pt0` is a special pattern term; it can be switched between **standard pattern term** and **transitional pattern term**.

- Standard pattern terms

  The standard pattern terms detect the **state** of a signal (either 0 or 1) in contrast to transitional patterns, which detect the change of a signal.

  If a standard pattern term queries multiple signals, all signals are combined via logical AND.

  To allow an easy trigger on any protocol combination, the pattern recognition of protocol attributes is aligned with the associated data transfer.

- Transitional pattern term

  The transitional pattern term detects **state changes** of signals. If it queries multiple signals, all signals are combined via logical OR.

  A transitional pattern term can be used for an efficient storage qualification when samples are to be taken only on *changes* of relevant signals.

Functions Overview

Programming Options

Programming pattern terms allows the following options:

- To specify a pattern term, use *BestPattSet*.

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

- To set compare patterns for trace memory control, use *BestTracePattPropSet*.

Example

Task Program the following three pattern terms:

- Detection of transactions on video memory for triggering:

  `pt0 = "b_state==3\h && AD32==b8xxx\h"`

  This makes pattern term `pt0` sensitive to address phases (`b_state==3\h`) and sensitive to signals on the address/data lines in the address space between `b8000\h` and `b8FFF\h`.

- Filtering of waits from stored data:

  `pt1 = "b_state==7\h"`

  This sets up pattern term `pt1` to detect data transfers. Inverted `pt1` (`!pt1`) can then be used to filter waits. This condition should be used as a storage qualifier.

- Detecting the end of a data transfer:

  `pt2 = "b_state==1"`

  This makes pattern term `pt2` sensitive to idles and thus to the end of the data transfer.

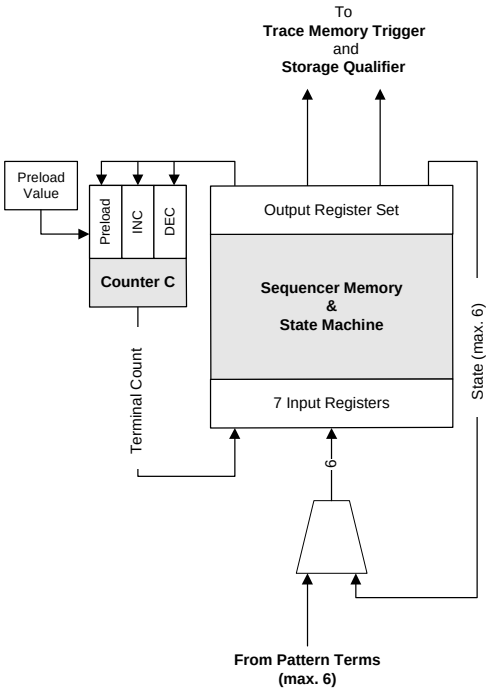
Implementation

```
err=BestPattSet(handle, \
                B_PATT_TERM_0, \
                "b_state==3\h && AD32==0b8xxx\h"); C(err);
err=BestPattSet(handle, \
                B_PATT_TERM_1, \
                "b_state==7\h"); C(err);
err=BestPattSet(handle, \
                B_PATT_TERM_2, \
                "b_state==1\h"); C(err);
```

Sequencer Programming

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

Representive of all sequencers, the figure below shows the trace memory trigger sequencer. The only difference to other sequencers is its output: the trace memory trigger signal and the storage qualifier signal.



All sequencers provide an internal memory and state machine, and a 32-bit feedback counter C. The statemachine controls the operation of the sequencer. The sequencer has 7 input registers. One of the registers is used for the terminal count of the sequencer's own feedback counter. The remaining 6 registers can be used for input from pattern terms and for state feedback from the sequencer output. A maximum of $2^5 = 32$ states is the practical limit—because at least one pattern term is always needed.

Setting up the Sequencer

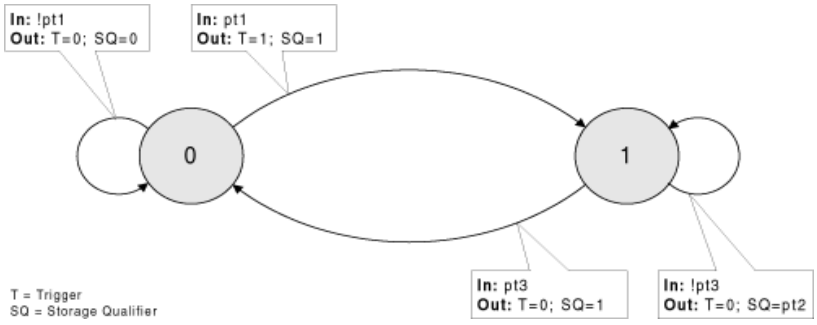
Setting up a sequencer requires the following steps:

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 the Pattern Terms" on page 52*.

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 may contain up to 256 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 counter enable condition

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

– Feedback counter preload conditions

  Output conditions to set the feedback counter 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 an excerpt from a sequencer description table.

Transient No.	Current State	Next State	Transition Condition	Output Conditions
0	0	0	!pt0	Depends on the sequencer
1	0	1	pt0	
2	1	1	!pt2	
3	1	0	pt2	

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.

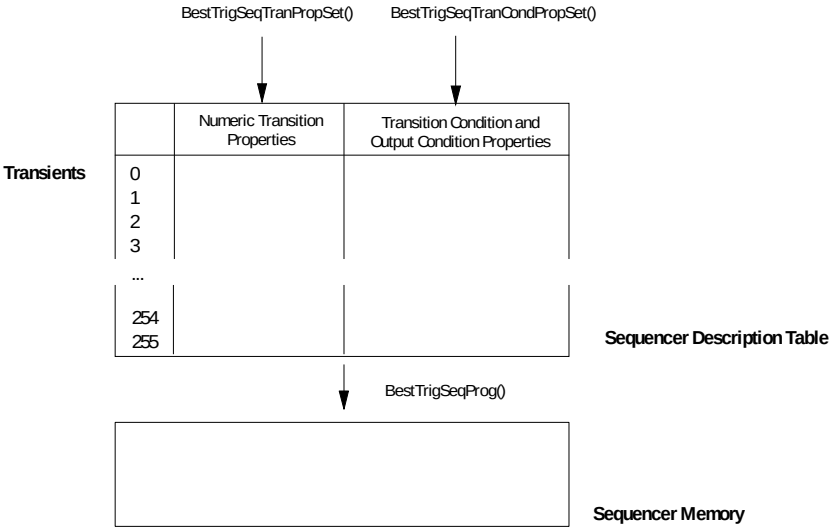
– Clock n+2: `tc` input to sequencer is asserted.

There may be additional sequence states and transitions required to get the desired sequencer behavior. See `<install_dir>\samles\gui\mwi_not8.cli` for an example.

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).

Functions Overview

The following figure gives an overview of the sequencer memory programming model (pattern terms and trigger position counter are not considered).



Programming Steps Programming the sequencer requires the following steps:

1. Set the preload value of the feedback counter.

   Each sequencer is equipped with a preloadable **feedback counter**. It can be decremented or loaded, enabling you to specify how often a sequence must occur before an output signal is set. Its output “tc” (terminal count) becomes 1 if the counter contains a value of 0xFFFFFFFF (–1).

   Use *BestTrigSeqGenPropSet*.

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

   Use *BestTrigSeqPropDefaultSet*.

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

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

Use *BestTrigSeqTranPropSet*.

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

- transition condition
- conditions to decrement and preload the feedback counter
- trigger condition and storage qualifier condition (only required for programming the trace memory trigger sequencer)

  All conditions (transition, trigger, storage qualifier, counter enable) 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 terms (`pt0`, `pt1`, ...) and the terminal count (`tc`) of the feedback counter C.

  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 *BestTrigSeqTranCondPropSet*.

5 Write the sequencer description table to the sequencer memory.

Use *BestTrigSeqProg*.

Example

Task

As an example, the trace memory trigger sequencer is programmed to do the following:

1. Capture any state until a transfer to the video memory address space occurs.
2. After this event, all data should be captured without waits until the data transfer is complete.

Pattern Terms

For this sequence, the following patterns need to be detected and are therefore assigned to pattern terms:

- `pt0 = "b_state==3\h && AD32==b8xxx\h"`

  Pattern term `pt0` detects transactions on video memory.

- `pt1 = "b_state==7\h"`

  Pattern term `pt1` filters waits.

- `pt2 = "b_state==1\h"`

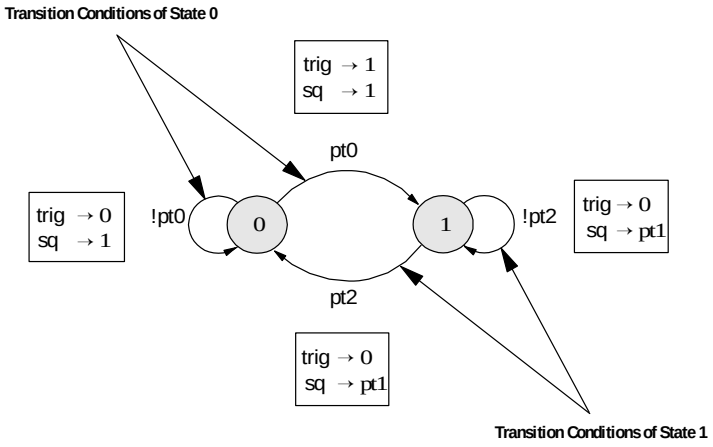
  Pattern term `pt2` detects the end of the transfer.

Building a State Diagram

The following figure shows the state flow for the example.

NOTE

In this example, feedback counters are not considered.



The example state diagram shows two states 0 and 1. Two transition conditions are used for state 0: `pt0` and `!pt0`. State 0 represents the state before the first access to video memory. At this time all bus states are stored in the trace memory (storage qualifier set to 1, no filtering).

The sequencer remains in state 0 until pattern term `pt0` turns true, that is, until the first access to video memory. When this event is detected, the sequencer switches to state 1 and sets the trigger signal to 1 (true).

The sequencer remains in state 1 until the bus goes idle, or as long as pattern term `pt2` is false. The trigger position counter starts to count down, and the storage qualifier is set to `pt1` to filter out waits. If waits occur, timestamp information is stored instead of them.

If the trigger position counter does not expire before `pt0` turns to 0, the sequencer switches back to state 0 as soon as the end of the data phase is detected (`pt2` turns true). In this state all data lines will be sampled until the trace memory is full.

The example results in the following sequencer description table:

Transient No.	State	Next State	Transition Condition	Trigger Condition (Output)	Storage Qualifier Condition (Output)	Description
0	0	0	!pt0	0	1	Before trigger event
1	0	1	pt0	1	1	Trigger event occurred (write to video memory)
2	1	1	!pt2	0	pt1	Sample without waits
3	1	0	pt2	0	pt1	Transfer completed

NOTE In the table above, only columns that need to be programmed for the example are shown. The columns “Feedback Counter Enable Condition” and “Feedback Counter Preload Condition” are skipped for clearness.

The sequencer starts in state 0. It observes the transition conditions of this state and sets the output conditions (trigger, storage qualifier and feedback counter count enable) according to the transition condition of the state that is true. (If none of them are true, the sequencer remains in the same state.)

In the example, this state is represented by transient 0 while the transition condition `pt0`: while the transition condition “transfer to video memory” does not become true the storage qualifier condition is set to 1: all states are sampled into trace memory.

If condition `pt0` becomes true, transient 1 is valid: the sequencer sets the trigger condition to 1 and moves to state 1 (“next state”).

Now the transition conditions of state 1 (transients 2 and 3) are observed and the storage qualifier condition is set to `pt1`, which filters waits. If condition `pt2`, end of data transfer, becomes true (transient 3), the sequencer switches back to state 0 (“next state”).

Implementation

The following C program fragment shows the programming for the example. It refers to the trace memory trigger programming. For other sequencers, similar commands are available.

```
/* Initialize the trace memory trigger sequencer description table.*/
err=BestTrigSeqPropDefaultSet(handle); C(err);

/* Preloads the trace memory trigger sequencer.*/
err=BestTrigSeqGenPropSet(handle, \
                          B_TRIGSEQGEN_CTRC_PREL, \
                          32); C(err);


/* Initialize and set up transient 0. */
err=BestTrigSeqTranPropDefaultSet(handle, 0); C(err);
err=BestTrigSeqTranPropSet(handle, 0, B_TRIGSEQ_STATE, 0); C(err);
err=BestTrigSeqTranPropSet(handle, 0, B_TRIGSEQ_NEXTSTATE, 0);
C(err);
err=BestTrigSeqTranCondPropSet(handle, \
                               0, B_TRIGSEQ_XCOND, "!pt0"); C(err);

err=BestTrigSeqTranCondPropSet(handle, \
                               0, B_TRIGSEQ_TRIGCOND, "0"); C(err);

err=BestTrigSeqTranCondPropSet(handle, \
                               0, B_TRIGSEQ_SQCOND, "1"); C(err);


/* Initialize and set up transient 1. */
err=BestTrigSeqTranPropDefaultSet(handle, 1); C(err);
err=BestTrigSeqTranPropSet(handle, 1, B_TRIGSEQ_STATE, 0); C(err);
err=BestTrigSeqTranPropSet(handle, 1, B_TRIGSEQ_NEXTSTATE, 1);
C(err);
err=BestTrigSeqTranCondPropSet(handle, \
                               1, B_TRIGSEQ_XCOND, "pt0"); C(err);
err=BestTrigSeqTranCondPropSet(handle, \
                               1, B_TRIGSEQ_TRIGCOND, "1"); C(err);

err=BestTrigSeqTranCondPropSet(handle, \
                               1, B_TRIGSEQ_SQCOND, "1"); C(err);


/* Initialize and set up transient 2. */
err=BestTrigSeqTranPropDefaultSet(handle, 2); C(err);
err=BestTrigSeqTranPropSet(handle, 2, B_TRIGSEQ_STATE, 1); C(err);
err=BestTrigSeqTranPropSet(handle, 2, B_TRIGSEQ_NEXTSTATE, 1);
C(err);
err=BestTrigSeqTranCondPropSet(handle, \
                               2, B_TRIGSEQ_XCOND, "!pt2"); C(err);
err=BestTrigSeqTranCondPropSet(handle, \
                               2, B_TRIGSEQ_TRIGCOND, "0"); C(err);
err=BestTrigSeqTranCondPropSet(handle, \
                               2, B_TRIGSEQ_SQCOND, "pt1"); C(err);
```

```
/* Initialize and set up transient 3. */
err=BestTrigSeqTranPropDefaultSet(handle, 3); C(err);
err=BestTrigSeqTranPropSet(handle, 3, B_TRIGSEQ_STATE, 1); C(err);
err=BestTrigSeqTranPropSet(handle, 3, B_TRIGSEQ_NEXTSTATE, 0);
C(err);
err=BestTrigSeqTranCondPropSet(handle, \
                               3, B_TRIGSEQ_XCOND, "pt2"); C(err);
err=BestTrigSeqTranCondPropSet(handle, \
                               3, B_TRIGSEQ_TRIGCOND, "0"); C(err);
err=BestTrigSeqTranCondPropSet(handle, \
                               3, B_TRIGSEQ_SQCOND, "pt1"); C(err);


/* Write the sequencer description table to the sequencer memory.
The transition conditions are checked for consistency. */
err=BestTrigSeqProg(handle); C(err);
```

Performance Measurement Programming

The testcard features eight performance measures, which 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.

For an overview of a performance measurement, refer to "*Operation Principles*" in the *Agilent E2925B PCI Analyzer User's Guide*.

Performance measurement can be divided into the following steps:

1. Setting up pattern terms and sequencers.

   For more information, refer to *"Programming the Pattern Terms" on page 52* and *"Sequencer Programming" on page 55*.

2. Programming the sequencer for performance measurement.

   For this purpose, the C-API provides an own function set. See *"Functions Overview" on page 65*.

3. Running the measurement and viewing the results.

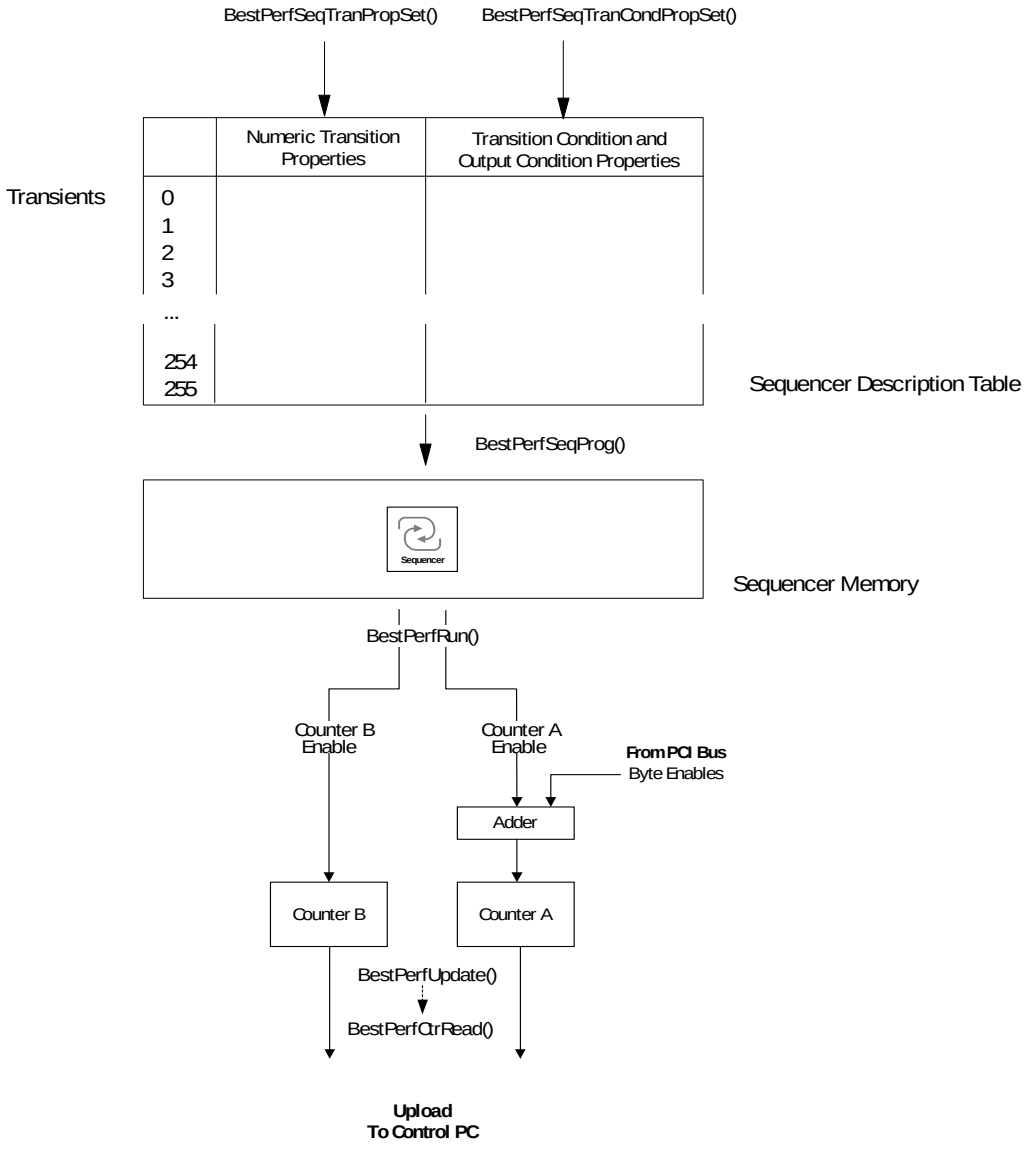
   For this purpose, the measures must be periodically updated, read and the desired values (for example, efficiency) must be computed.

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

   See *"Example" on page 67*.

Functions Overview

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



Programming Steps Programming performance measurements requires the following steps:

1 Set the preload value for feedback counter C.

Use *BestPerfSeqGenPropSet*.

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

2 Set all properties in the performance sequencer description table to default values.

Use *BestPerfSeqPropDefaultSet*.

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

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

Use *BestPerfSeqTranPropSet*.

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

– transition condition

– conditions to increment nominator or denominator counter

– conditions to decrement or preload the feedback 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 referring to pattern terms (`pt0`, `pt1`, ...) and the terminal count (`tc`) of the feedback counter C.

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

Use *BestPerfSeqTranCondPropSet*.

5 Write the sequencer description table to the sequencer memory.

Use *BestPerfSeqProg*.

6 To run the measurement, start the counters.

Use *BestPerfRun*.

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

Use *BestPerfStatusGet*.

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

Use *BestPerfUpdate* and *BestPerfCtrRead*.

9 You can stop the performance measurement manually.

Use *BestPerfStop*.

Example

Task To explain how to program a performance measure, an example trace measurement is used. It is intended to measure the following:

- the average bus non-idle time in percent
- the efficiency in percent

Pattern Terms The pattern terms are to be set up as follows:

- `pt4= "b_state==7\h"`

  To measure the amount of transferred data by using IRDY# and TRDY#, `pt4` is sensitive to data transfers.

NOTE IRDY# and FRAME# cannot be combined directly, because OR is not allowed in the pattern term equation. Refer to "*Standard Pattern Term Operators*" in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

- `pt5 = "b_state==1\h"`

  To measure the non-idle time of the bus, pattern term `pt5` is sensitive to idle times.

The **sequencer** must be programmed to remain in state 0 only, incrementing counter A and B if their enable conditions are true. In the example, counter A and counter B of performance measure 0 are used. The sequencer description table must be set up as follows:

Transient No.	State	Next State	Transition Condition	Counter A Enable Condition (Output)	Counter B Enable Condition (Output)
0	0	0	1	pt4	!pt5

NOTE In the table above, only columns are shown that need to be programmed for the example. The columns "Feedback Counter Enable Condition", and "Feedback Counter Preload Condition" are skipped for clarity.

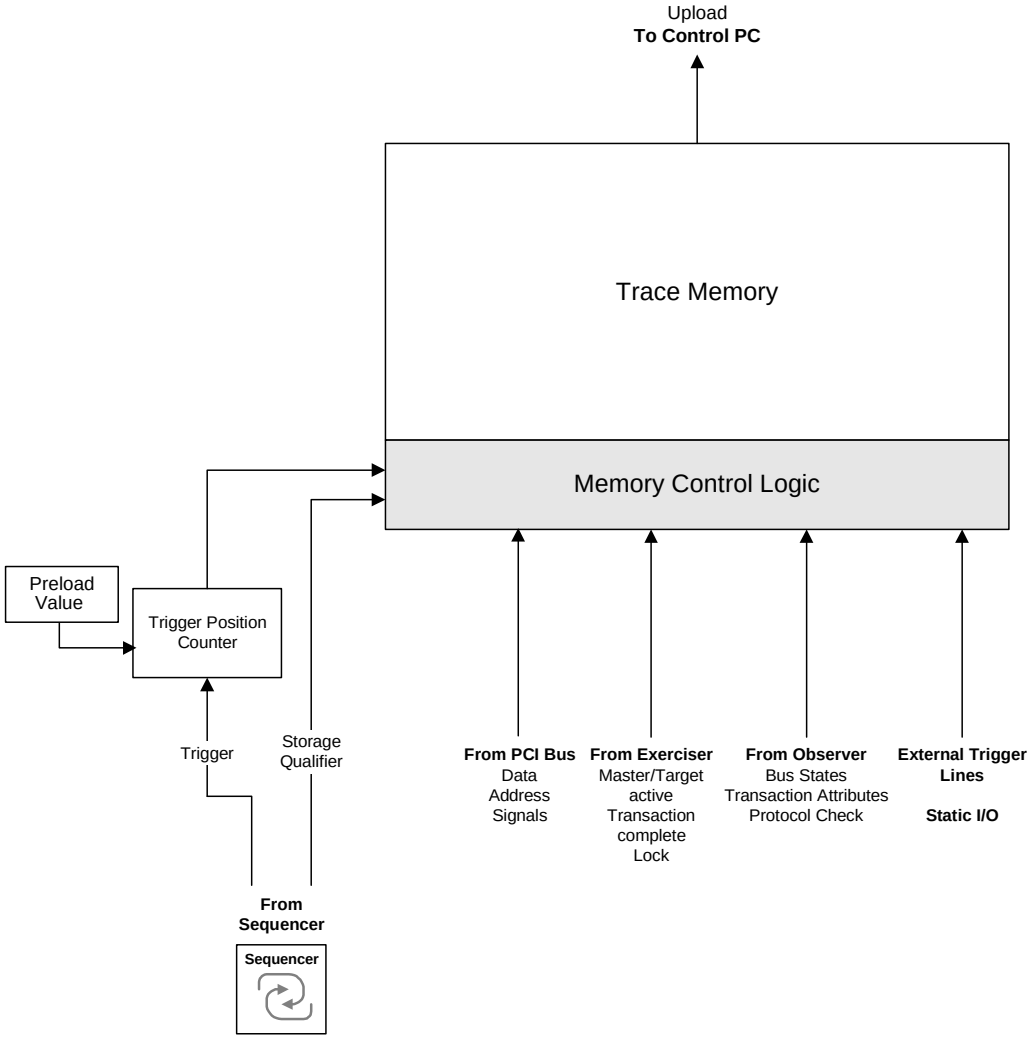
Implementation

```
/* Implement the pattern terms and the sequencer as described
above.*/

/* Set the generic properties of performance measure 0 to default
values to set the feedback counter value to 0. */
err=BestPerfGenPropDefaultSet(handle,B_PERFMEAS_0); C(err);

/* Program the pattern terms pt4, pt5 and pt6. These are the
pattern terms used in the sequencer conditions. */
err=BestPattSet(handle,B_PATT_TERM_4,"b_state==7\h"); C(err);
err=BestPattSet(handle,B_PATT_TERM_5,"b_state==1\h"); C(err);

/* Set up the sequencer description table of performance measure 0
to its default values. */
err=BestPerfSeqPropDefaultSet(handle,B_PERFMEAS_0); C(err);

/* Initialize transient 0. */
err=BestPerfSeqTranPropDefaultSet(handle,B_PERFMEAS_0, 0); C(err);

/* Set the counter A of performance measure 0 to count the number
of transferred bytes (byte enables). */
err=BestPerfGenPropSet( handle, \
                        B_PERFMEAS_0, \
                        B_PERFGEN_CAMODE, \
                        B_CAMODE_INCRBYTEN ); C(err);

/* Set up transient 0. */
err = BestPerfSeqTranPropSet( handle, \
                              B_PERFMEAS_0, \
                              0, \
                              B_PERFSEQ_STATE, \
                              0);C(err);

err = BestPerfSeqTranPropSet( handle, \
                              B_PERFMEAS_0, \
                              0, \
                              B_PERFSEQ_NEXTSTATE, \
                              0); C(err);

err = BestPerfSeqTranCondPropSet( handle, \
                                  B_PERFMEAS_0, \
                                  0, \
                                  B_PERFSEQ_XCOND, \
                                  "1"); C(err);

err = BestPerfSeqTranCondPropSet( handle, \
                                  B_PERFMEAS_0, \
                                  0, \
                                  B_PERFSEQ_CA_EN, \
                                  "pt4"); C(err);

err = BestPerfSeqTranCondPropSet( handle, \
                                  B_PERFMEAS_0, \
                                  0, \
                                  B_PERFSEQ_CB_EN, \
                                  "!pt5"); C(err);
```

```
/* Program the sequencer of performance measure 0. */
err=BestPerfSeqProg(handle,B_PERFMEAS_0); C(err);

/* Start the counters. */
err=BestPerfRun(handle); C(err);

/* The following loop periodically updates the measures, reads
them, and computes the non-idle time and efficiency values. To
compute the percentage of non-idle time, the total amount of time
is taken from the reference counter value.*/

while(1)
{
   err=BestPerfUpdate(handle); C(err);

/* Read the counter values*/
   err=BestPerfCtrRead( handle, \
                        B_PERFMEAS_0, \
                        B_PERFCTR_A, \
                        &counter_a); C(err);

   err=BestPerfCtrRead( handle, \
                        B_PERFMEAS_0, \
                        B_PERFCTR_B, \
                        &counter_b); C(err);

   err=BestPerfCtrRead( handle, \
                        B_PERFMEAS_0, \
                        B_REFCTR, \
                        &ref_counter); C(err);

/ * Compute the required data, non-idle time and overall transfer
efficiency. */
   non_idle   = ((float)counter_b / (float)ref_counter) * 100;
   efficiency = ((float)counter_a / ((float)counter_b * 4)) * 100;

/* Print the results to the standard output once in 1000 ms. */
   printf("Bus Non-Idle: %2.2f%% Efficiency: %2.2f%%    \r", \
           non_idle,efficiency);
   sleep(1000);
} /* end while */
return(0);
```

Trace Memory Programming

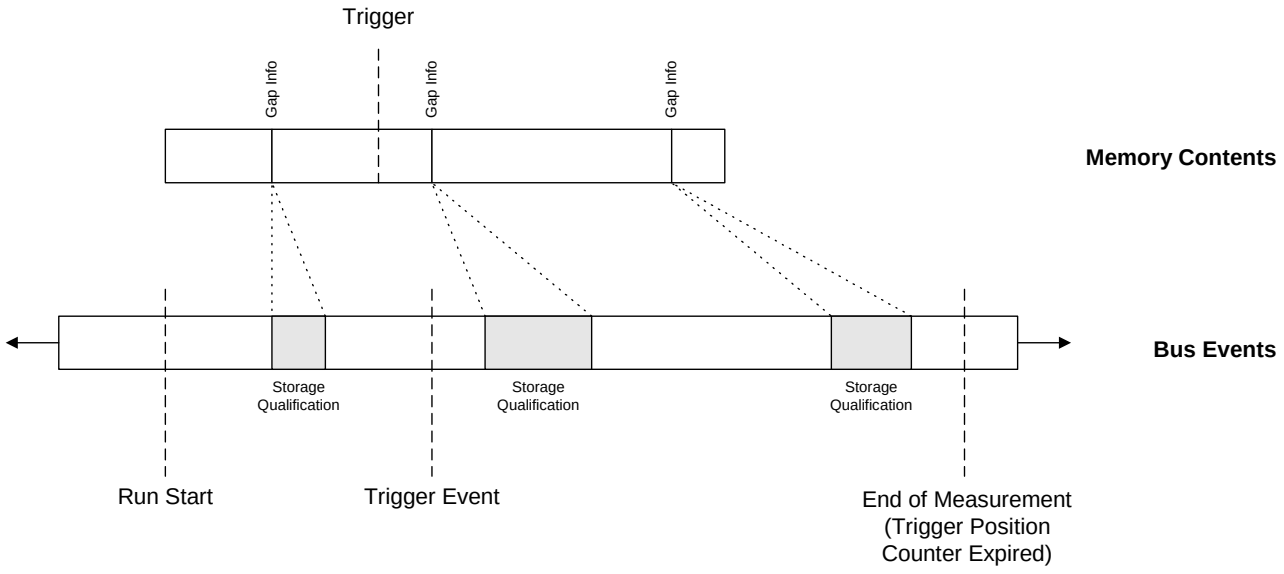
The figure below gives an overview of the components of the trace memory:



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 the Pattern Terms" on page 52* and *"Sequencer Programming" on page 55*.

Functions Overview

Programming Steps

Programming the trace memory requires the following steps:

1. Set up the trace memory, preload the trigger position counter and set the gap mode.

   Use *BestTracePropSet*.

2. Program the sequencer and the pattern terms.

3. Run the test by starting the trigger sequencer and the trace memory.

   Use *BestTraceRun*.

   The course of the test can be monitored:

   – by polling the *trace status register*, or

   – by watching the LEDs on the board.

   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 the run should be manually stopped, use *BestTraceStop*.

   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. Upload the trace memory depending on an occurred trigger event, store the trace memory line number where the trigger event has occurred.

   Use *BestTraceStatusGet*.

6. Read trace data from the trace memory, begin with the line where the trigger event has occurred.

   Use *BestTraceDataGet*.

7. Analyze the data lines for certain signals, proceed as follows:

   – Determine the position and size of the desired signals within the data line.

   Use *BestTraceBitPosGet*.

   – Terminate the connection.

   – Print out signal status information.

Example

NOTE In this example, it is assumed that pattern terms and sequencer are programmed as described in *"Programming the Pattern Terms" on page 52* and *"Sequencer Programming" on page 55*. Furthermore, it is assumed that the preload value of the *trigger position counter* is left at its default value, so that the trigger event will be found in the center of the trace memory.

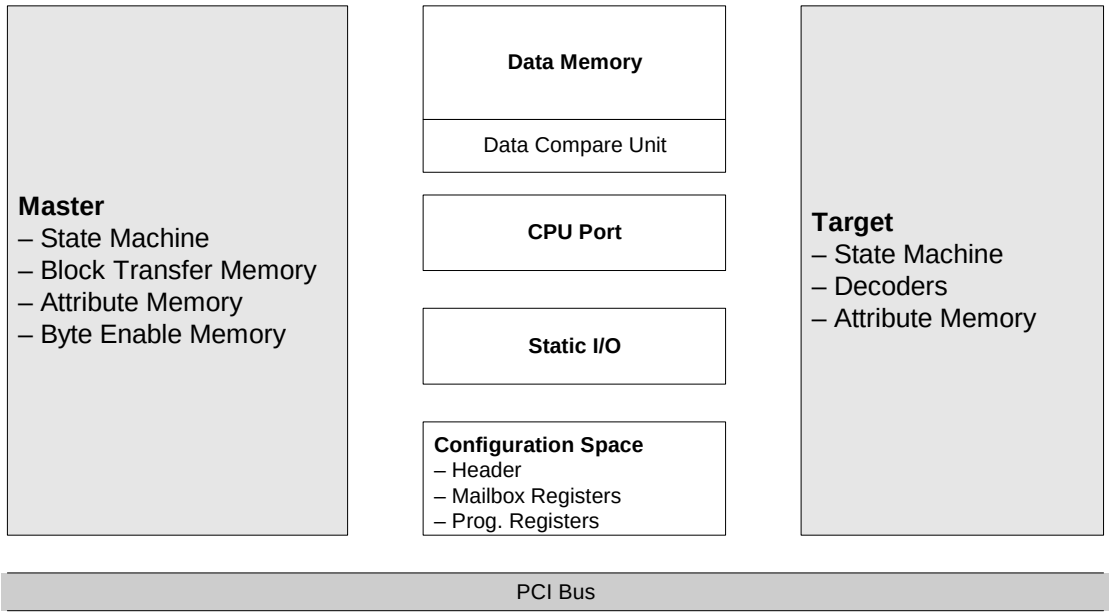
```
/* Sets the gap mode to store exhaustive gap information (normal
gap mode). */
err=BestTracePropSet( handle, B_TRC_ANALYZER_MODE, B_PERFORMANCE);

/* Programming the sequencer description table and the pattern
terms goes in here. */

/* … */

/* Start the trigger sequencer and trace memory. */
err=BestTraceRun(handle); C(err);

/* Poll the trace status register. */
    do
    {
      err=BestTraceStatusGet(handle, B_TRC_STAT, &status_reg);
    } while (status_reg & 0x01);

/ * Stop the run. */
err=BestTraceStop(handle); C(err);

/* Read the trace memory line number where the trigger event has
occurred and stores it in the "triggerline" variable. */
err=BestTraceStatusGet(handle, \
                       B_TRC_TRIGPOINT, triggerline); C(err);

/* Write 32 lines of trace data are written to the variable
"tracedata" (you need to assign a buffer for the trace data
previously). The program reads them from the trace memory,
beginning with the line stored in "triggerline" from the previous
call. */
err=BestTraceDataGet(handle, triggerline, 32, tracedata); C(err);

/* Interprete the captured data.
Note that you need to assign buffer for the variables, this is not
considered in the example code fragment. */
```

```
/* Determine the position and size of the desired signals within
the data line. */
err = BestTraceBitPosGet(handle, B_SIG_AD32, &ad32_pos, &ad32_len);
C(err);
err = BestTraceBitPosGet(handle, B_SIG_CBE3_0, &cbe_pos, &cbe_len);
C(err);
err = BestTraceBitPosGet(handle, B_SIG_FRAME, &frame_pos,
&frame_len); C(err);
err = BestTraceBitPosGet(handle, B_SIG_IRDY, &irdy_pos, &irdy_len);
C(err);
err = BestTraceBitPosGet(handle, B_SIG_TRDY, &trdy_pos, &trdy_len);
C(err);
err = BestTraceBitPosGet(handle, B_SIG_DEVSEL, &devsel_pos,
&devsel_len); C(err);
err = BestTraceBitPosGet(handle, B_SIG_STOP, &stop_pos, &stop_len);
C(err);

/* Terminate the connection. */
err=BestDisconnect(handle); C(err);
err=BestClose(handle); C(err);

/* Print out signal status information. */
printf("Line #   \tAD\tC/BE\tCTRL\n");

for (i = 0;i <= (lines-upload_start)*bytes_per_line/4;
i+=(bytes_per_line/4))
{
   printf("%06d   \t%08lx \t %1lx \t%c%c%c%c%c\n",
      disp_start,
      tptr[i + ad32_pos/32],
      (tptr[i + cbe_pos/32]>>(cbe_pos%32)) & ((1<<cbe_len)-1),
      (((tptr[i + frame_pos/32]>>(frame_pos%32)) & 1) ? ' ' : 'F'),
//FRAME
      (((tptr[i + irdy_pos/32]>>(irdy_pos%32)) & 1) ? ' ' : 'I'),
//IRDY
      (((tptr[i + trdy_pos/32]>>(trdy_pos%32)) & 1) ? ' ' : 'T'),
//TRDY
      (((tptr[i + devsel_pos/32]>>(devsel_pos%32)) & 1) ? ' ' :
'D'), //DEVSEL
      (((tptr[i + stop_pos/32]>>(stop_pos%32)) & 1) ? ' ' : 'S'));
//STOP
   disp_start++;
}
return (0);
```

Programming the Exerciser

The exerciser provides a master and a target and some resources that are shared by both. All these can be controlled by functions of the C-API.

The following figure shows the components of the testcard.



NOTE If master and target of the testcard are set up to transfer data from one to the other, the target has no access to data memory. If the master writes to the target, data memory will remain unchanged. If the master reads from target, it will receive dummy data.

The following sections explain how to program the components of the testcard's exerciser:

- *"Reading from and Writing to the Memories" on page 77* describes the principles of programming the memories.
- *"Programming the Exerciser as a Master Device" on page 80* describes how to program the data transfer.
- *"Programming the Exerciser as a Target Device" on page 105*

- *"Data Memory and Compare Unit Programming" on page 142*
- *"Host Access Programming" on page 144*
- *"Interrupt Programming" on page 146* describes how to generate a PCI interrupt with the exerciser.
- *"Built-In Test Programming" on page 147* describes prepared tests, which can be set up and be performed with less programming effort.

Reading from and Writing to the Memories

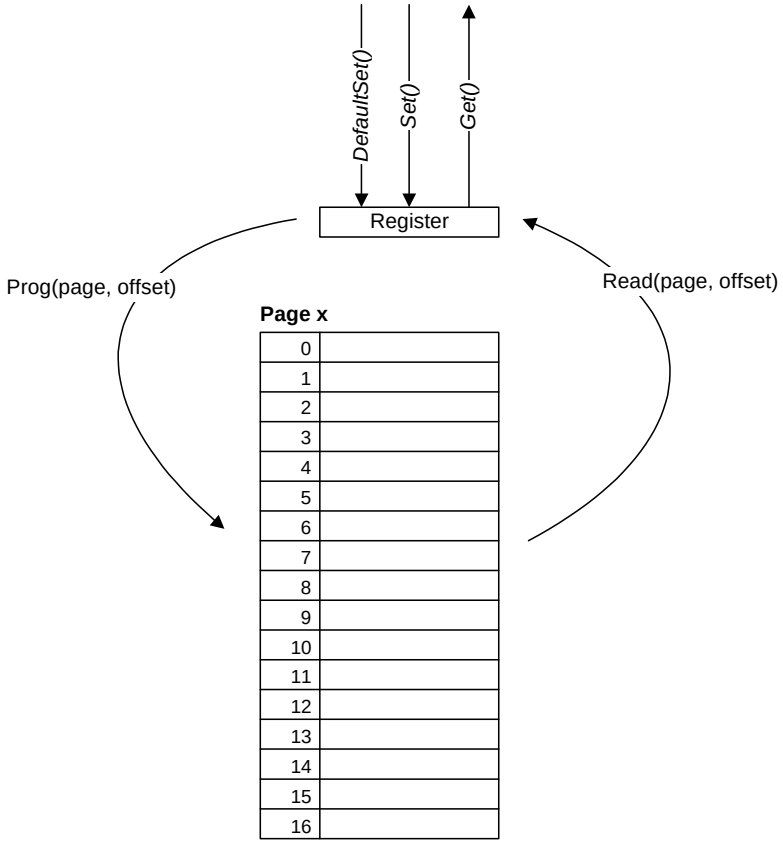
When programming the memories, they are accessed via a preparation register. This register can hold all properties of one memory line.

To program the memory, you can:

- Program this register first and then download it to a memory line.
- Read a memory line into the register, read and modify its values and download the modified registers to a memory line.

Preparation Register

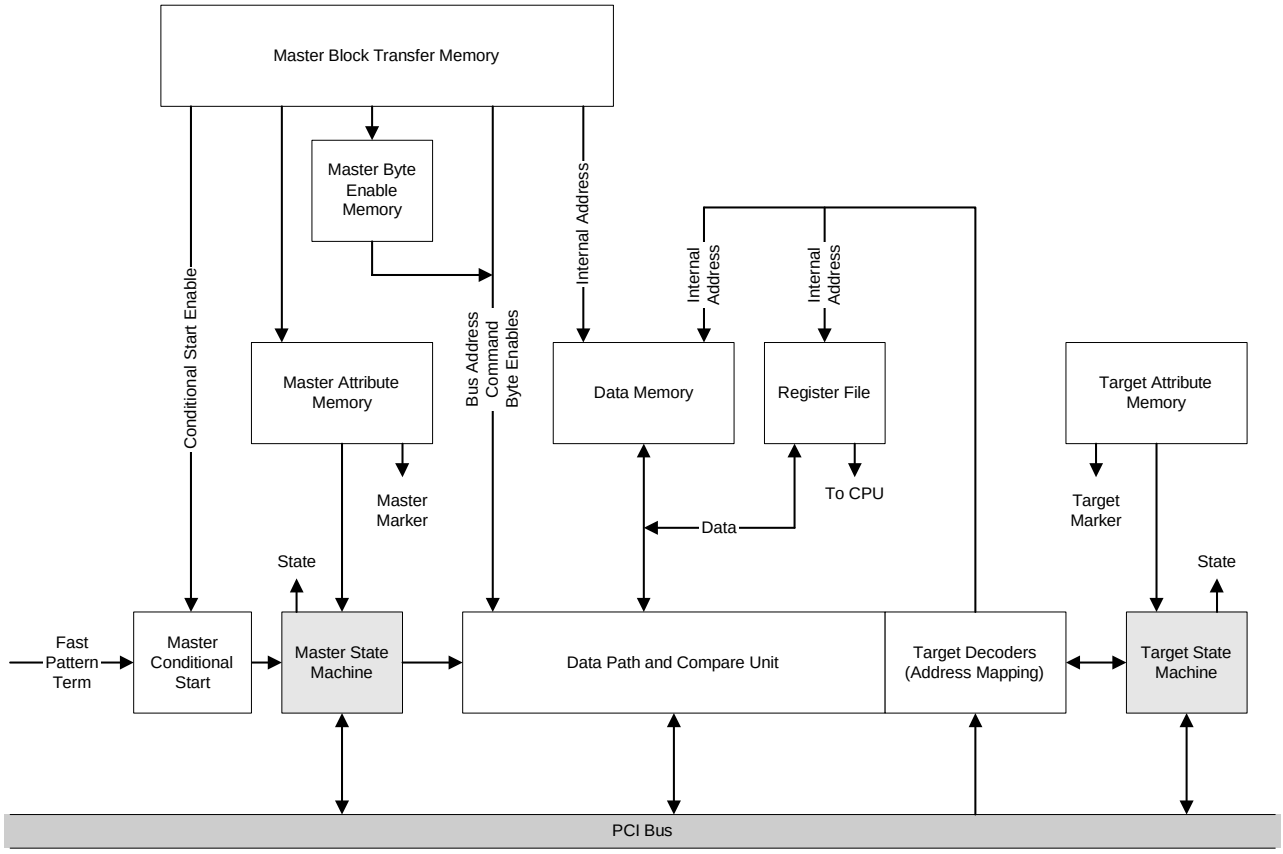
The following figure shows the principle:



When programming a memory line with the content of the preparation register, a page pointer and a line pointer are passed with the referring command or function. These pointers are used to identify the exact memory line for reading and writing between the memory and preparation register.

Exerciser Block Diagram

The following figure shows a detailed block diagram of the testcard's exerciser. Experience PCI exerciser users can use this diagram to see exactly how the testcard's exerciser functions.



This figure gives a precise picture of the exerciser components:

- the components of the **master** are shown on the left side
- the components of the **target** are shown on the right side
- the **shared resources** are shown between them:
  - on-board data memory
  - register files
  - compare unit

The central component of both the master and the target is its **state machine**. This controls the testcard's behavior as a master or target and is controlled by programmable memories (for example, protocol behavior is controlled by the attribute memories).

The state machines provide a "State" signal, the attribute memories provide a "Marker". Both, "State" signal and "Marker" are **exerciser outputs** and can be used by the testcard's analyzer, for example to trigger the trace memory when "Marker" or "State" signals show particular values.

The "Fast Pattern Term" is **exerciser input** from the testcard's analyzer and is used for the conditional start mode of the master state machine.

Programming the Exerciser as a Master Device

To program the testcard's exerciser as a master device means programming the testcard to initiate data transfers via PCI bus either to a target device under test, or to the testcard's own 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 master properties.

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

   See *"Programming Generic Master Properties" on page 82*.

2. The master transactions to be performed.

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

   See *"Master Block Transfer Memory Programming" on page 85*.

3. The protocol attributes to be used with the transactions.

   Protocol attributes determine the behavior during the various phases of each transaction, for example, whether errors should be signaled during address phases or how many waits should be inserted during a data phase. This information is located in the master attribute memory.

   See *"Master Attribute Memory Programming" on page 89*.

4. The data to be used for the transactions.

   For this purpose, the data memory can be used as a data resource by the master.

   See *"Data Memory and Compare Unit Programming" on page 142*.

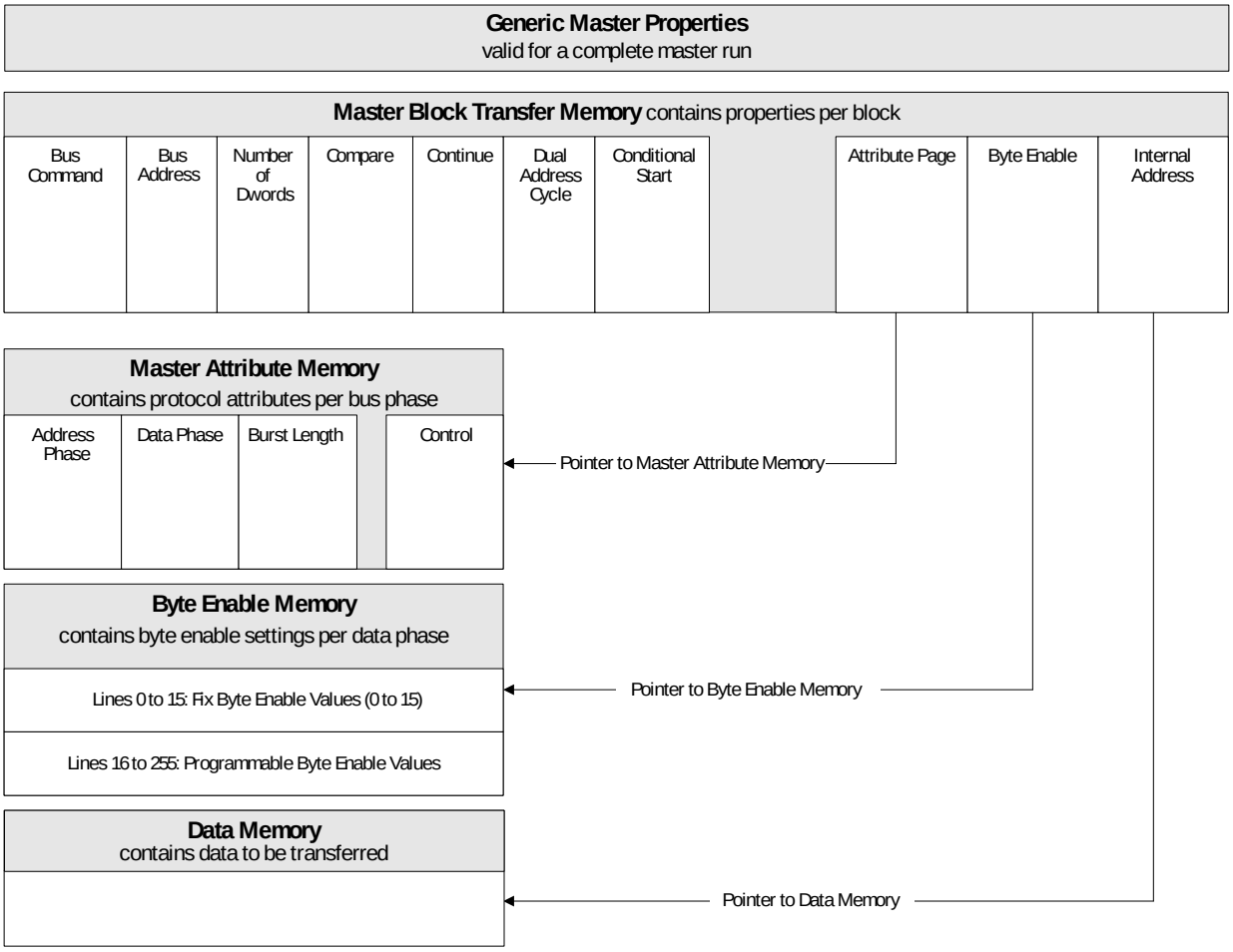
5. The number of byte enables to be set in a data phase.

   See *"Byte Enable Memory Programming" on page 101*.

6. The way of running the master.

   See *"Master Run" on page 103*.

Master Memories The following figure shows the various memories needed to initiate transactions on the PCI bus and gives an overview of their contents.



Programming Generic Master Properties

You can program the following generic master properties:

- **Run Mode, Trigger and Delay Counter**

  The run mode determines whether the master should start immediately, conditionally after a trigger event, or after a programmable delay.

- **Repeat Mode**

  The master operation can be executed only once, or run repeatedly.

- **Master Attribute Pointer Mode**

  This property determines at what time the internal master attribute memory pointer is reset to start an attribute memory page (after each block, after each block page, or not at all).

- **Invert Data**

  This property determines whether or not outgoing data bytes that are masked by byte enables are inverted (refer to *"Data Memory and Compare Unit Programming" on page 142*).

The following properties refer to entries in the testcard's configuration space:

- **Latency Timer**

  The testcard's latency timer can be preset. It can be shut off for testing outside of the PCI specification.

- **MWI Mode**

  The testcard's "Memory Write and Invalidate" bit can be set or reset.

- **"Master Enable" Bit**

  The testcard's "Master Enable" bit can be set or reset.

- **"Fast Back-to-Back" Bit**

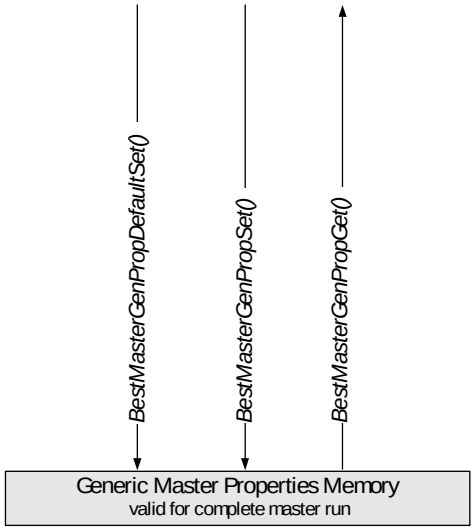
  The testcard's "Fast Back-to-Back" bit can be set or reset.

- **Cacheline Size**

  The cacheline size that is assumed for the system under test can be determined.

Functions Overview

The following figure shows the functions used to program the generic master properties memory.



The generic master properties are not programmed using a preparation register. The generic master property memory is simply programmed by …`Set()` and …`Get()` functions, so that no further description is needed.

Example

Task Program an immediate, single master transfer. Although the mode is immediate, the transactions should start only after the run function has been called and the arbiter has granted the bus to the testcard.

Implementation

```
/* Set all generic master properties to defaults and enable the
master. */
err=BestMasterGenPropDefaultSet(handle); C(err);
err=BestMasterGenPropSet(handle, B_MGEN_MASTERENABLE,1);C(err);

/* Program the master to start after a trigger pattern has been
detected on the bus, and then to run indefinitely.*/

/* Setting the run mode to "wait on delay". */
err=BestMasterGenPropSet(handle, \
                  B_MGEN_RUNMODE, \
                  B_RUNMODE_WONDELAY);C(err);

/* Set a trigger pattern. */
err=BestMasterCondStartPattSet( handle, \
                  "b_state=3\\h & CBE3_0=7\\h & AD32=b8xxx\\h"); \
                  C(err);

/* Set the repeat mode to "infinite". */
err=BestMasterGenPropSet(handle, \
                  B_MGEN_REPEATMODE, \
                  B_REPEATMODE_INFINITE);C(err);

printf ("generic run property infinite programmed\n");
```

Master Block Transfer Memory Programming

Memory Contents

All properties needed for a bus transfer, for example, the addresses from and to where data is to be transferred, must be programmed to the master block transfer memory. The memory cannot be accessed directly, so reading from and writing to the memory is performed via the preparation register (see *"Programming the Exerciser as a Master Device" on page 80*). The master block transfer memory also holds pointers to the other memories, which control the exerciser behavior per bus phase (master attribute memory, byte enable memory, data memory).

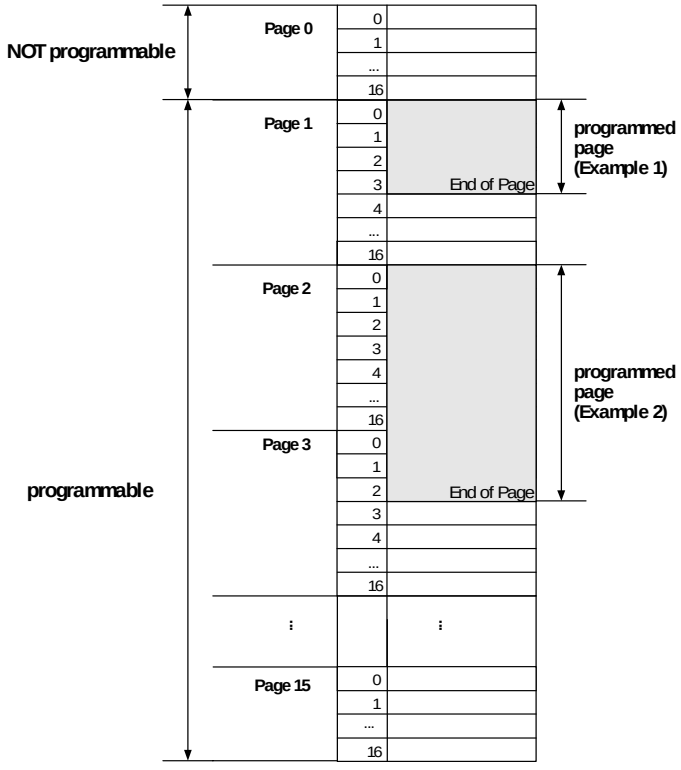
For a detailed description of all programmable properties, please refer to the "*Transaction Properties*" in the *Agilent E2925B Opt. 300 PCI Exerciser User's Guide*.

Before you start programming the properties to the memory, you should have a good knowledge of the memory design.

Master Block Transfer Memory Design

The memory is organized in 16 pages: page 0 contains default values, pages 1 to 15 are programmable. Each page contains 17 lines, each line represents **one** block transfer. This results in 255 programmable lines.

The following figure shows how the master block transfer memory is designed and gives examples for programmed pages.



Structure of Programmed Pages

The first line of a programmed page is used as an entry point. The last line of the page must contain an "end of page" information (EOP) instead of block transfer properties. The lines after EOP cannot be programmed.

For example, **Page 1** contains 3 lines with block transfer properties and one EOP line. The remaining lines of this page must be left blank. Programming can be continued on page 2 only.

Concatenated Pages

If more than 17 blocks are to be transferred, you can write these lines by crossing the border to the next page. The next page is then concatenated, and you lose one entry point.

Following the example, **Pages 2 and 3** are concatenated pages and contain 19 blocks and 1 EOP line. Because the size of page 2 is exceeded, page 3 is no longer available for direct access except for initialization. If page 3 is accessed for initialization, page 2 will also be initialized.

The remaining lines (3 to 16) of page 3 must be left blank.

Running the Bock Transfers

When a master block page run is started with page 1 of the previous example, the lines of page 1 will be executed one after the other until the EOP is found. Then the run will usually stop.

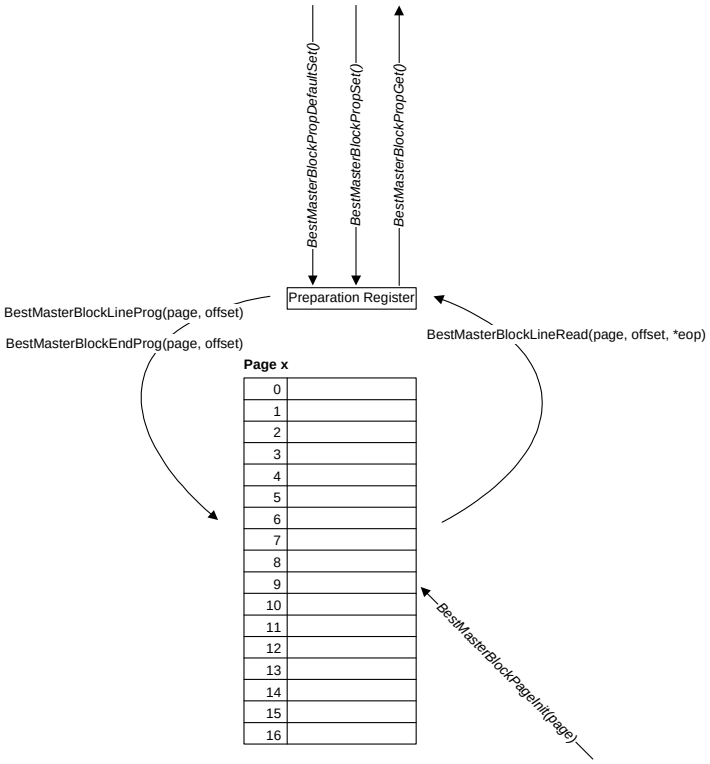
If the master block page run is started with page 2 of the previous example, the lines of page 2 and 3 would be executed in the same manner: one after the other until an EOP is found.

A master run started with page 3 would result in an error, as well as any programming access to a line on page 3 (for example, to change data in a line on page 3). The entries in page 3 are only available via page 2 using an offset greater than 16.

For more details in running the transfers, please refer to *"Master Run" on page 103*.

Functions Overview

The following figure shows the functions used to program the master block transfer memory.



Programming Steps

Programming master block transfers requires the following steps:

1. Initialize a page in the master block transfer memory.

   Use *BestMasterBlockPageInit*.

2. Set the preparation register to default values.

   Use *BestMasterBlockPropDefaultSet*.

3. Change block transfer properties in the preparation register as needed.

   Use *BestMasterBlockPropSet*.

4. Program a memory line with the content of the preparation register.

   Use *BestMasterBlockLineProg*.

5. Repeat steps 3 and 4 for each block you want to program in the block page.

6. Conclude the memory page with “end of page” (EOP).

   Use *BestMasterBlockEndProg*.

Repeat these steps for all the block transfer pages you need.

Example

Task Program a read of 40 dwords from memory address `0xb8e60` (video memory) to the testcard's internal memory address 0 using the master attributes in attribute memory page "MyAttrPage".

For programming an attribute memory page, see *"Master Attribute Memory Programming" on page 89*.

Implementation

```
/* Initialize the block page "MyBlockpage" */
err=BestMasterBlockPageInit(handle, MyBlockPage);C(err);
/* Set the preparation register to default values */
err=BestMasterBlockPropDefaultSet(handle);C(err);

/* Write to the preparation register: */
/* the starting bus address for the block transfer */
err=BestMasterBlockPropSet(handle,B_BLK_BUSADDR,0xb8e60);C(err);
err=BestMasterBlockPropSet(handle, \
                           B_BLK_INTADDR, \
                            0x00);C(err);
/* the read command */
err=BestMasterBlockPropSet(handle, \
                            B_BLK_BUSCMD, \
                            B_CMD_MEM_READ);C(err);
/* the number of 40 dwords */
err=BestMasterBlockPropSet(handle,B_BLK_NOFDWORDS,40);C(err);
/* the pointer to the master attribute page "MyAttrPage" */
err=BestMasterBlockPropSet(handle,B_BLK_ATTRPAGE,MyAttrPage);C(err);

/* Write the content of the preparation register to memory line 0
*/
err=BestMasterBlockLineProg(handle, MyBlockPage, 0);C(err);
/* Define memory line 1 to EOP */
err=BestMasterBlockEndProg(handle, MyBlockPage,1);C(err);
```

Master Attribute Memory Programming

Memory Contents

The master attribute memory contains information on how the blocks stored in the master block transfer memory should be transferred.

The settings of this memory do not affect the result of the data transfer. You can repeat transfers with **fixed master block transfer** settings but with **varying master attributes,** and then compare the transferred data with data stored in the testcard's internal memory. For example, the master attributes could be varied by a randomizer.

Basically, programming the master attribute memory is not different from programming the master block transfer memory. It is also designed like a table divided into pages. Therefore, it is recommended that you first read *"Master Block Transfer Memory Programming" on page 85*.

The following explanations describe the differences between the memories.

Memory Design

The master attribute memory can hold up to **256 lines**. It is organized in **64 pages** with **4 lines** each. Page 0 contains default values and is read-only.

NOTE

For compatibility with older C-API versions, there is also an 8-page by 32-line memory organization. This is also the default. To take advantage of the enhanced number of pages, change the master attribute page size with *BestExerciserGenPropSet* and store this as power-up property so that this setting will be used at every power-up. (This does not affect the principles described in the following.)

Each line represents one successful bus phase (this is different to the behavior of the target) and contains both address and data attributes. If a phase was unsuccessful (for example due to target retries), it will be repeated using the same line until successful completion.

Instead of an "end of page" line—as used in the master block transfer memory— this information is held in the loop bit. The loop needs to be set to "0" except in the last line. In this way, it is ensured that the exerciser cycles through the master attributes and returns to the beginning.

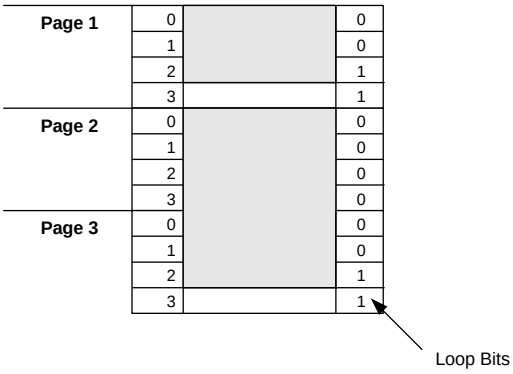
Building Loops in the Master Attribute Memory

The following figure shows how loops are built in the master attribute memory.

Line	Content	Loop
x	Attribute Set 0	0
x+1	Attribute Set 1	0
x+2	Attribute Set 2	1
x+3		1

Programming Concatenated Pages

The following figure shows how pages with more than 4 lines are to be programmed in concatenated pages. The figure is also used to explain how the exerciser works through the master attribute memory:



Page 1 contains 3 entries with master attributes. It is referred by the master block transfer property “Attribute Page” with value 1. The exerciser works through the lines until it finds the row with a loop bit set to “1”, and then restarts the page with the next phase. In this way it continues until the number of dwords specified in the master block transfer properties has been transferred, or the transactions must end due to other circumstances.

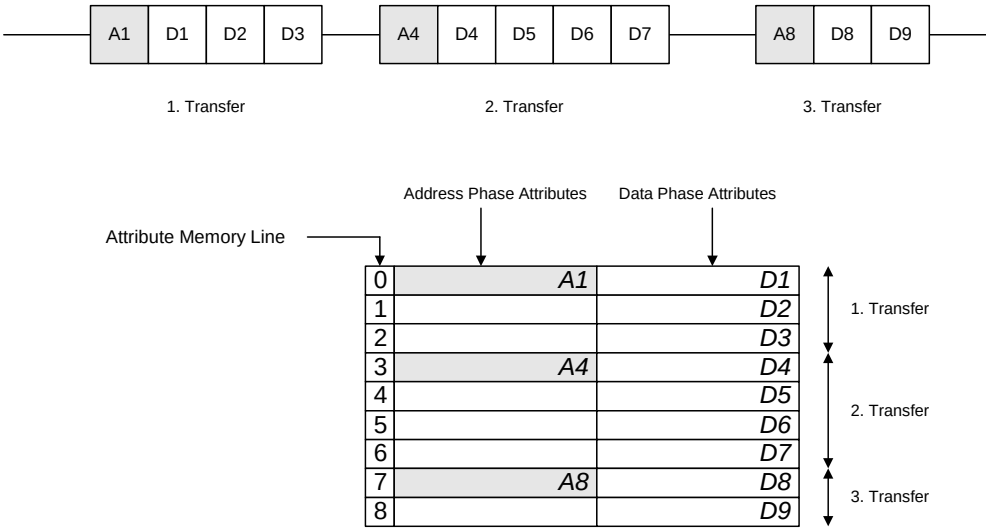
Line 3 in page 1 can be left on its default values. It is out of the loop and will not be executed.

Pages 2 and 3 are concatenated pages and contain 7 lines with master attributes, which are worked through by the exerciser when it refers page 2. A reference to page 3 would result in an error (except for initialization). Line 3 of page 3 can again be left as it is, because it is out of the loop and will not be executed.

NOTE There is no need to program an extra end of page identifier. The pages’ ends are identified by the loop bit.

Working through the Memory

The following figure shows how the attribute memory is worked through with each bus phase:



The figure shows three transfers, each one consisting of one address phase and several data phases. There is one column for the attributes for address phases, and another column for the attributes for data phases.

The attributes for one transfer must be programmed into consecutive lines. There is one line for each data phase. In each line

- the first column holds the address phase attributes (the same attributes for each line belonging to one transfer),
- the second column holds the data phase attributes for one data phase (different attributes for different data phases).

Normally only the address phase attributes in the first line for a transfer are used, those in the following lines are ignored. If, however, the master *must* continue with an address phase (for example, after a target retry that occurred within a transaction), it will use the address phase attributes from the current line.

To achieve a deterministic attribute behavior of the master, each block transfer must start with the beginning of a master attribute page. This can be controlled with transaction property `B_BLK_CONTATTR`. See "*b_blkproptype*" in the *C-API/PPR Reference*.

Master Attributes

This memory contains all properties of a bus phase during a transfer, such as how many master waits are to be inserted into a data phase or whether a parity error should be signaled during an address phase and so forth.

A master attribute memory line covers all types of master attributes:

- address phase attributes
- data phase attributes
- control attributes

The next memory line is selected after each *successful data phase*. The address phase attributes of a line are ignored during a data phase.

Address Phase Attributes

The exerciser uses the address phase attributes only during an address phase of a new transaction. Otherwise the address phase attributes are ignored. You can find all address phase attributes with their detailed descriptions in *"Address Phase Attributes (Master)"* in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

Data Phase Attributes

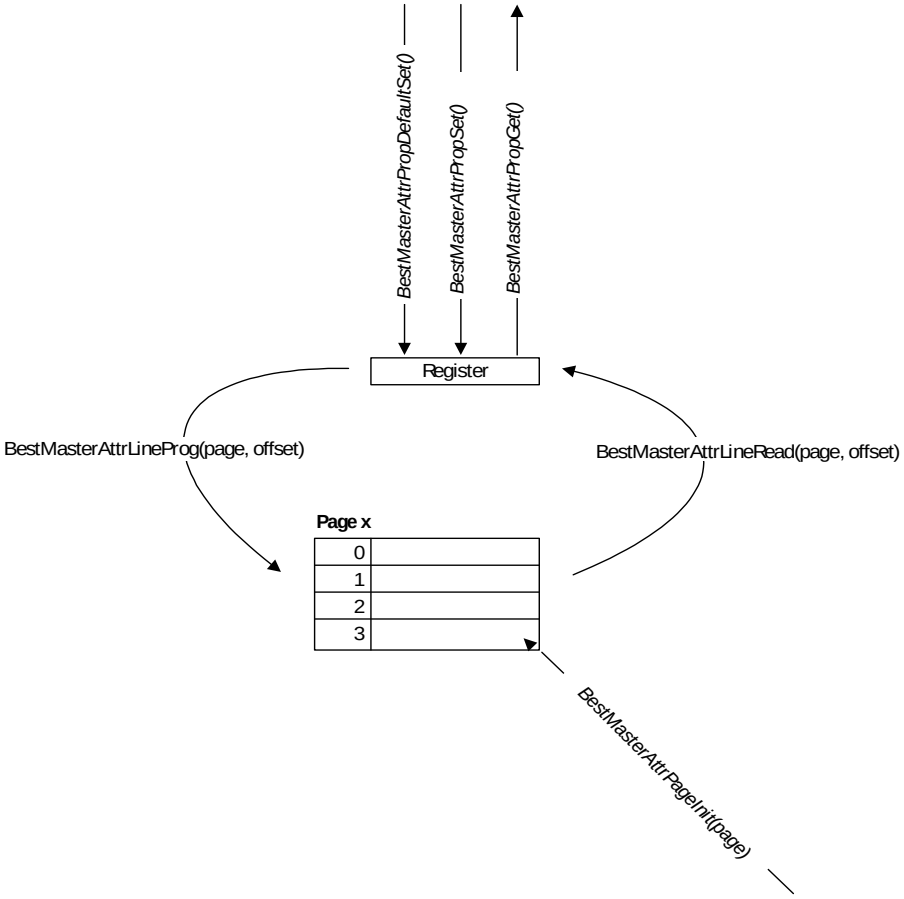
The exerciser uses the data phase attributes only during a data phase of a transaction. Otherwise the attributes are ignored. You can find all address phase attributes with their detailed descriptions in *"Data Phase Attributes (Master)"* in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

Control Attributes

The control attributes are used to determine how the exerciser works through the master attribute memory. The control attributes are valid for data and address attributes. You can find all address phase attributes with their detailed descriptions in *"Control Attributes (Master)"* in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

Functions Overview

The following figure shows the C functions used when programming master attribute pages:



Programming Steps

Programming master attributes requires the following steps:

1. Initialize a page in the master attribute memory.
   Use *BestMasterAttrPageInit*.
2. Set the preparation register to default values.
   Use *BestMasterAttrPropDefaultSet*.
3. Change attributes in the preparation register as needed.
   Use *BestMasterAttrPropSet*.
4. Program a memory line with the content of the preparation register.
   Use *BestMasterAttrLineProg*.
5. Repeat steps 3 and 4 for each line you want to program in the attribute page.

Repeat these steps for all the attribute pages you need.

Example

Implementation

The following lines program 3 attribute phases with an increasing number of wait states:

```
err=BestMasterAttrPageInit(handle,MyAttrPage);C(err);
err=BestMasterAttrPropDefaultSet(handle);C(err);

/* First Address or Data Phase */
err=BestMasterAttrPropSet(handle,B_M_WAITS,1);C(err);
err=BestMasterAttrLineProg(handle, MyAttrPage, 0);C(err);

/* Second Address or Data Phase */
err=BestMasterAttrPropSet(handle,B_M_WAITS,3);C(err);
err=BestMasterAttrLineProg(handle, MyAttrPage, 1);C(err);

/* Third Address or Data Phase, and goto first */
err=BestMasterAttrPropSet(handle,B_M_WAITS,5);C(err);
err=BestMasterAttrPropSet(handle,B_M_DOLOOP,1);C(err);
err=BestMasterAttrLineProg(handle, MyAttrPage, 2);C(err);
```

Master Attribute Group Programming

To achieve more sophisticated randomization opportunities, the master attributes are divided into groups, which can be varied against each other. For this purpose, the C-API provides an own function set controlling the attributes and the loop bit per group (see *"Functions Overview" on page 98*).

Attribute Groups

The following table shows which attribute is assigned to which group:

Group	Address Phase Attributes
MA0	Delay (Exerciser Idle)
MA1	Try "Fast Back-to-Back", Steps
MA2	Lock, 64-Bit Request, Release Request
MA3	Resume Delay
MA4	Wrong Parity Signalling, Parity and System Errors

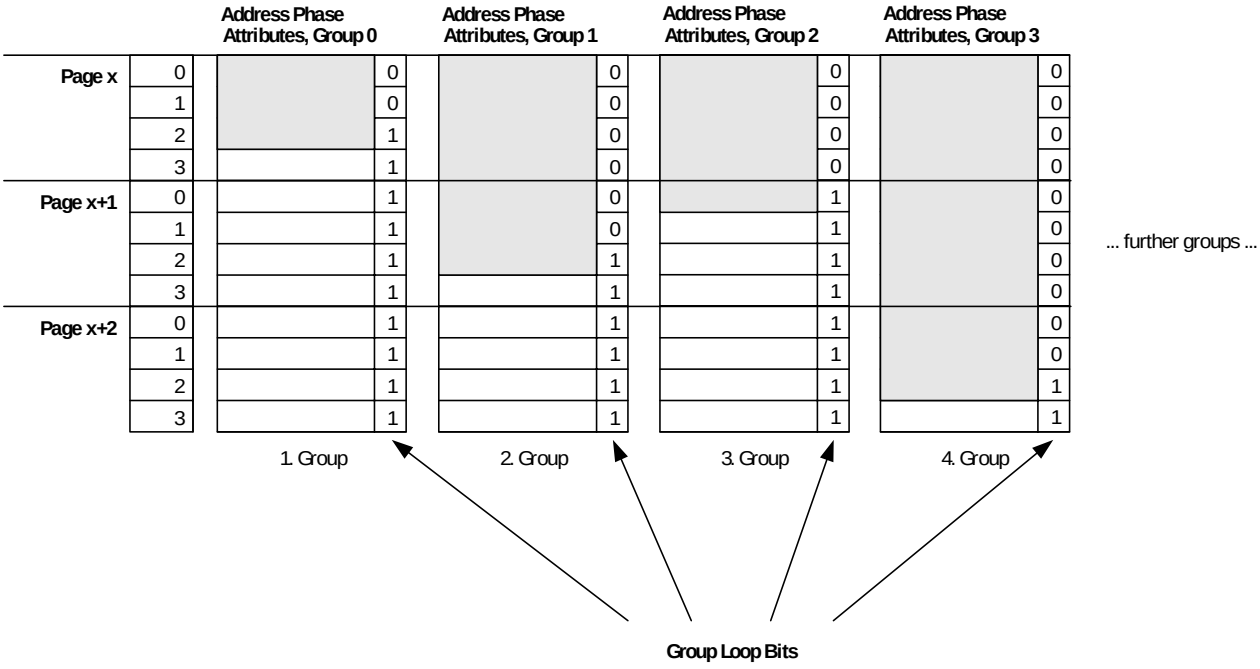
Group	Data Phase Attributes
MD0	Waits
MD1	Release Request, Parity and System Errors
MD2	Wrong Parity Signalling, Marker

Group	Control Attributes
ML	Last, Repeat

This assignment is fixed and cannot be programmed or otherwise changed.

Composing a Master Attribute Page

A master attribute page can be composed of particular group pages of different sizes. The following figure shows an example:



The figure shows four groups of page x:

- The first group is address phase attribute group 0.
- The second group is address phase attribute group 1.
- The third group is address phase attribute group 2.
- The fourth group is address phase attribute group 3.

In the figure, the remaining groups are skipped for clearness.

The preset page size of 4 lines is exceeded by all groups except the first one. The fourth group exceeds 8 lines. Therefore, the pages “x+1” and “x+2” are concatenated and also part of page x.

The fourth group provides the most lines—11—and thus determines the overall page size of page x. The next available page is page x+3.

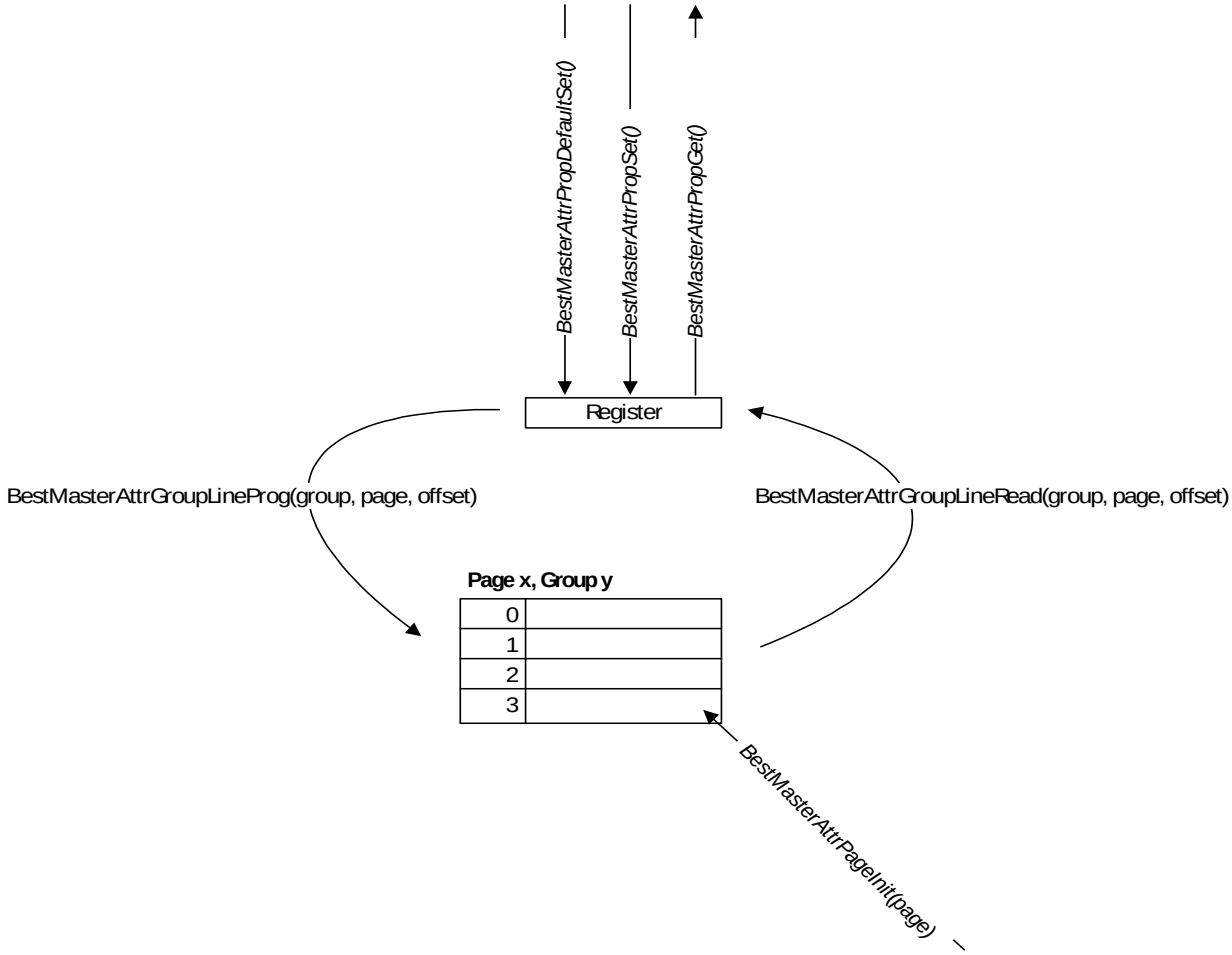
When the exerciser calls page x, the attributes of the groups are cycled through *independently* per transaction phase. This means:

- The first three phases use lines 0 to 2 of page x over all groups.
- Phase four uses line 3 of the second, third and fourth group—but line 0 of the first group: this group restarts at its page begin.
- The third group restarts in the sixth phase. At this moment, the first group will have cycled through the second time and is “standing” on line 2.
- The second group restarts in phase eight. At this moment, the first group is in line 1 and the third group is in line 2.
- The fourth group restarts in phase 12. At this moment, the first group is in line 2, the second is in line 4, the third is in line 1.

NOTE The group page sizes should always be prime numbers and should always differ to vary the attributes of the groups against each other.

Functions Overview

NOTE Within a page, the master attribute group functions are not allowed to be mixed with the non-group functions.



Programming Steps Programming master attribute groups requires the following steps:

1. Initialize a master attribute page. The page will be identified by its number.
   Use *BestMasterAttrPageInit*.
2. Set the preparation register to default values.
   Use *BestMasterAttrPropDefaultSet*.
3. Change attributes in the preparation register as needed for one group.
   Use *BestMasterAttrPropSet*.

4 Program an attribute memory group line with the content of the preparation register.
Use *BestMasterAttrGroupLineProg*.

5 Repeat steps 3 and 4 for all groups you want to program in one line.

6 Repeat steps 3, 4 and 5 for each line (address/data phase) you want to program in the attribute page.

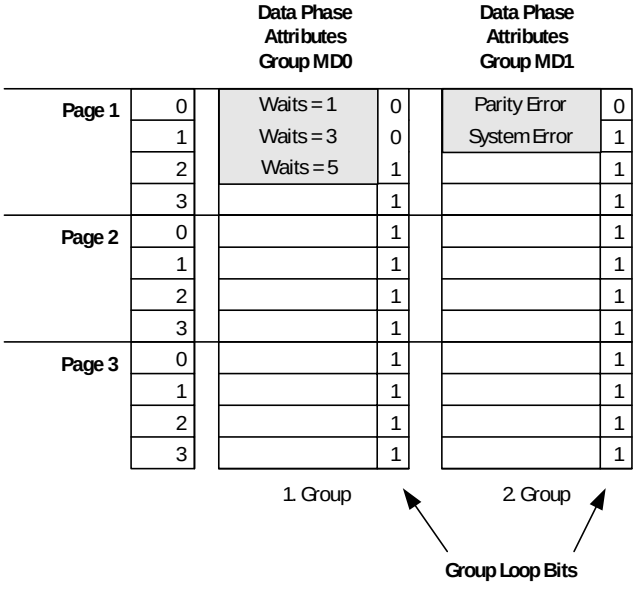
For each line within the group, leave the loop bit at 0, but in the last line of the group set this bit to 1. Use attribute `B_M_LOOP`.

7 Repeat these steps for all the attribute pages you need.

8 Select the page to be used (per session).
Use *BestMasterAttrPageSelect*.

Example

Task Program the following master attribute group pages:



Implemention

```
pg_num=1;

offset=0;

/* Initialize attribute page 1. */
err = BestMasterAttrPageInit(handle, 0x01); C(err);

/* Set the preparation register to default values. */
err = BestMasterAttrPropDefaultSet(handle); C(err);

/* Set the attributes for line 0 for group MD0 (master data0) in
the preparation register. */
err = BestMasterAttrPropSet(handle, B_M_WAITS, 0); C(err);

/* Program the attribute group MD0 to set waits in line 0 of the
attribute memory. */
err = BestMasterAttrGroupLineProg(handle, B_MATTR_GRP_MD0, pg_num,
offset); C(err);

/* Set the attributes for line 0 for group MD1 (master data1) in
the preparation register. */
err = BestMasterAttrPropSet(handle, B_M_DOLOOP, 0); C(err);
err = BestMasterAttrPropSet(handle, B_M_DSERR, 1);

/* Program the attribute group MD1 (master data1) to set a system
error in line 0 of the attribute memory. */
err = BestMasterAttrGroupLineProg(handle, B_MATTR_GRP_MD1, pg_num,
offset); C(err);

offset++;

/* Set the attributes for line 1 for group MD0 (master data0) in
the preparation register. */
err = BestMasterAttrPropSet(handle, B_M_WAITS, 3); C(err);

/* Program the attribute group MD0 (master data0) to set waits in
line 1 of the attribute memory. */

err = BestMasterAttrGroupLineProg(handle, B_MATTR_GRP_MD0, pg_num,
offset); C(err);

/* Set the attributes for line 1 for group MD1 (master data1) in
the preparation register. */

err = BestMasterAttrPropSet(handle, B_M_DOLOOP, 0); C(err);
err = BestMasterAttrPropSet(handle, B_M_DPERR, 1);

// Program the attribute group MD1 (master data1) to set a parity
error in line 1 of the attribute memory. */

err = BestMasterAttrGroupLineProg(handle, B_MATTR_GRP_MD1, pg_num,
offset); C(err);

offset++;
```

```
/* Set the attributes for line 2 for group MD0 (master data0) in
the preparation register. (Set the loop bit for group TD0). */
err = BestMasterAttrPropSet(handle, B_M_WAITS, 5); C(err);
err = BestMasterAttrPropSet(handle, B_M_DOLOOP, 1); C(err);

/* Program the attribute group TD0 (master data0) to set waits in
line 2 of the attribute memory. */
err = BestMasterAttrGroupLineProg(handle, B_MATTR_GRP_MD0, pg_num,
offset); C(err);

/* Select this attribute page to be used. */

err = BestMasterAttrPageSelect(handle, 0x01); C(err);
```

Byte Enable Memory Programming

Memory Content

The byte enable memory contains the information on which byte enables are to be set in a data phase. This can decrease the number of dwords to be transferred. The exerciser will consider a dword as exercised, even though it has been "masked" out due to byte enable setting.

Memory Design

The byte enable memory holds up to **256** lines of **8-bit-values**. Of these, **240** are **freely programmable**. The first 16 lines are fixed, their upper and lower 4-bit values hold the line numbers (for compatibility reasons). The following table shows how the byte enable memory is designed:

Line	Upper 4 bit	Lower 4 bit	Values are
0	0\h	0\h	fixed
1	1\h	1\h	
2	2\h	2\h	
...	...	...	
13	D\h	D\h	
14	E\h	E\h	
15	F\h	F\h	
16 ... 256	...		programmable

NOTE

The compare unit (see *"Data Memory and Compare Unit Programming" on page 142*) also considers byte enable settings during comparison. Therefore, bytes that are not transferred due to byte enable settings will not be compared.

The “Byte Enable” is a master block transfer property and points to a line in the byte enable memory. Furthermore, the byte enables can per transaction be controlled in the following ways (controlled by the master block property “Variable Byte Enable”):

- **Fixed byte enables**

  Each phase of the transaction uses the same byte enable setting, as specified in the master block transfer properties.

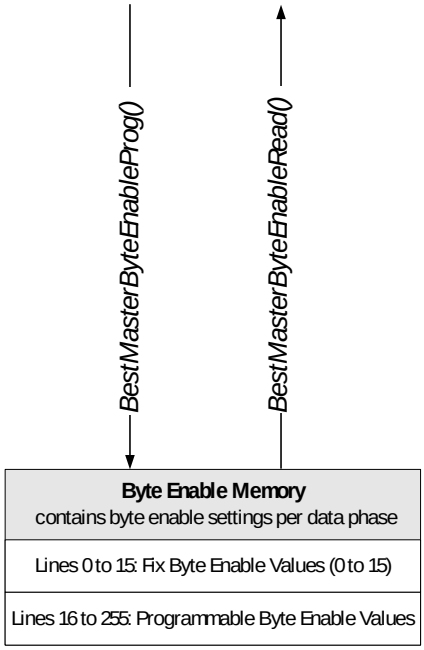
- **Variable byte enables**

  The byte enables vary with each data phase. The sequence starts with the byte enable pointed to by the byte enable pointer in the master block transfer properties. It then works through the lines of the byte enable memory. Variable byte enables can only be used with blocks with less than or equal 240 transfers.

This—pointer and variable byte enable on/off—is summarized by the term “Byte Enable Control” in *“Master Block Transfer Memory Programming” on page 85*.

Functions Overview

The following figure shows the functions used to program the byte enable memory.



The byte enables are not programmed using a preparation register. The byte enable memory is simply programmed by …`Prog()` and …`Read()` functions, so that no further description is needed.

Master Run

For a master run, the memories need to be programmed according to your requirements. When programming is finished, the exerciser can start a master run.

Two Ways for a Master Run

There are two ways to start a master run:

- **Block Page Run**

  The run is started with a master block transfer memory page. The lines of the page are worked through until the end of page (EOP) is found.

- **Block Run**

  Only one block is executed. The run ends when the block is completely worked through or when the run must be aborted.

Steps for Performing a Master Run

Basically the exerciser performs a master run as follows:

1. If a start condition is specified by the corresponding block transfer property, the exerciser waits until the start condition is met.
2. The exerciser requests the bus from the arbiter and waits for the grant.
3. After the bus is granted, the exerciser performs the transactions, which is controlled by the master state machine:
   - The exerciser picks the PCI bus address from the current master block transfer properties and drives it onto the bus.
   - The exerciser waits for the target's ready signal and then starts the transfer.
   - The exerciser performs the transfer phase by phase, stepping through and using the attributes stored in the master attribute memory.
4. The transaction is complete after the last dword is transferred, as specified in the master block transfer property "Number of Dwords".
5. A *block run* is then finished, unless the "Repeat" property has been set. In this case, the block would be repeated until it is externally stopped.
6. A *block page* run would call the next block and proceed in the same way as with the first block until the end of page is found. If the "Repeat" property has been set, the block page will be repeated until it is externally stopped.

Functions Overview

The Agilent E2925B testcard's programming interface provides functions for a master run. The available functions and their usage is shown by describing the programming steps.

Programming Steps

Programming the master run requires the following steps:

1 Start the transactions on the bus.

Use *BestMasterBlockPageRun*.

NOTE To run only one block specified by the current settings in the block page preparation register, use *BestMasterBlockRun*. This is useful, for example, for testing purposes.

The functions returns immediately after initiating the run.

2 Detect the end of the run by polling the status register.

Use *BestStatusRegGet*.

NOTE If the master does not stop on its own, it can be stopped by *BestMasterStop*.

Example

Task Run a block page in the master block transfer memory completely.

Implementation

```
/* Running block page 1 */
BestMasterBlockPageRun(handle,1);

/* Polling the status register to detect the end of the run */
   do
      {
      err=BestStatusRegGet(handle, &statusreg); C(err);
      }
   while(statusreg & 0x01);
```

Programming the Exerciser as a Target Device

To program the testcard's exerciser as a target device means programming the testcard to react to transactions initiated from a master device. This test can be used to test the functionality of a master device on the bus.

Programming the target needs the following steps:

- The generic target properties must be set.
- The target decoder must be set up.
- The configuration space must be modified.
- The protocol attributes must be programmed.
- The settings must be stored as power-up defaults.

Target Operation

In contrast to the master, a target cannot perform a "target run" as it has a passive behavior. After setting up and enabling the decoders and base address registers, the exerciser is able to react to accesses of master devices.

However, before running a test with the current decoder settings, the system under test and the testcard can first be switched off and on to test the start-up behavior during the **configuration phase**.

In a system with BIOS, the configuration phase is performed with the following steps:

1. When starting up, the testcard uses the programmed *power-up properties*. They determine the behavior of the decoders, for example, how to react on configuration access of the BIOS.
2. To scan for PCI topology and for connected PCI devices, the BIOS systematically asserts *IDSEL lines* of all slots within the system. If the configuration decoder of the exerciser is programmed to react on its IDSEL, it will assert DEVSEL# and thus signal to the BIOS that it is present.
3. The BIOS writes and reads to and from the configuration space to program the testcard, so that it can react to the correct memory and I/O requests.

4. After configuration, the BIOS *enables* the programmed decoders by setting the enable bits in the configuration space.

However, during configuration the exerciser can react to IDSEL a multiple number of times, and it can also react to IDSELs that are directed to other devices, and thus *pretend* to be other devices.

NOTE This allows emulation of bridges, bus topologies, multiple devices, and multifunction devices.

To do this, multiple standard decoders of the testcard must be set up with "config" behavior and be connected to different internal resources (for example, different partitions of data memory). The resources must be set up in such a way that they emulate the different configuration spaces and control the testcard's behavior when the testcard pretends to be another device.

Regardless of whether the exerciser emulates one or multiple devices, and whether the configuration space(s) are programmed by BIOS or by a program running on the testcard, after the configuration phase, the exerciser will claim **memory** and **I/O transactions** directed to the address ranges entered in its configuration space.

During such a transaction, the exerciser proceeds as follows:

1. It asserts DEVSEL# to signal to the requesting master that it claims the transaction, and analyzes the transaction for a command and address within its range.
2. According to the direction given by the command, it
   - either drives data taken from the connected *internal resource* onto the PCI bus
   - or reads data from the PCI bus and directs them to the connected *internal resource*.

   If the internal resource is the data memory or compare unit, the exerciser can use the **target attributes** stored in the target attribute memory.
3. The master will signal to the target when the transaction has been completed. The exerciser deasserts DEVSEL# and turns to idle state until the next transaction occurs.

If the testcard is plugged into a system with BIOS, settings such as decoder base addresses and size should not be changed while the system is running. Therefore, the C-API provides functions to store decoder settings as power-up defaults (refer to *"Power-Up and Reset Control" on page 37*.)

Programming Generic Target Properties

The generic target properties determine the general target behavior of the testcard. They are used to:

- Direct the testcard to use the attributes as specified in the attribute memory for transactions.
- Disable all memory decoders during and after programming to ensure that they do not decode accesses from an external source (the BIOS enables them after configuration during start up).

The Agilent E2925B testcard allows you to program the following generic properties:

- **Run Mode**

  Transfers to and from data memory can be performed with protocol attributes varying from phase to phase. The protocol attributes to be applied per bus phase are stored in the target attribute memory in one line per phase. The lines are organized in pages.

  The run mode property determines whether the target restarts with the first line of the attribute memory page with each transfer, or whether it always continues with the next line.

- **Enabling/disabling Expansion ROM decoder, Memory decoder, and I/O decoder**

  These properties enable or disable decoders. They are used to switch the decoders on or off, and set the corresponding enable/disable bit in the configuration space of the testcard. This prevents the BIOS or other masters from accessing testcard resources and decoders when they are programmed by the control software.

  The decoders can be enabled by the BIOS as soon as the configuration phase has finished, or—in a testing environment without BIOS—by means of the C-API.

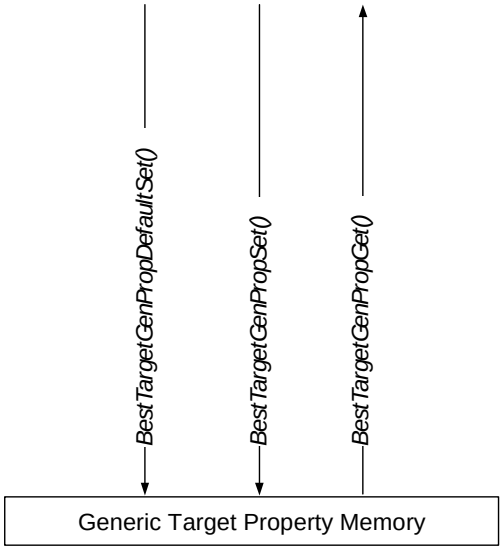
- **Fast Back-to-Back Capability**

  This property enables or disables the testcard's capability to perform Fast Back-to-Back cycles. It sets the corresponding enable/disable bit in the command register of configuration space of the testcard.

For more information about the generic target properties, refer to "*b_targetgenproptype*" in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

Functions Overview

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



The generic master properties are not programmed using a preparation register. The generic master property memory is simply programmed by …`Set()` and …`Get()` functions, so that no further description is needed.

Example

Task Program the target to continue always with the next line of the attribute memory page with each transfer and enable the memory decoder.

Implementation

```
err=BestTargetGenPropSet(handle, \
                         B_TGEN_RUNMODE, \
                         B_RUNMODE_SEQUENTIAL);C(err);
err=BestTargetGenPropSet(handle, \
                         B_TGEN_MEMSPACE, 0);C(err);
```

Programming the Target Decoder Properties Memory

Memory Contents

The target decoder properties memory contains information about the decoders. There is one memory line space per decoder for all target decoder properties, however, not all properties are used by each decode.

Decoders of the Testcard

The Agilent E2925B testcard provides the following decoder:

- 6 standard decoders
- Expansion ROM decoder
- Configuration decoder
- Full configuration decoder (type 1 configuration decoder)
- Subtractive decoder

Because full configuration decoder and subtractive decoder can only be controlled via the C-API, they are described in the following. For more information about the remaining decoders, refer to "*Target Decoder Properties*" in the *Agilent E2925B Opt. 320 PCI Exerciser User's Guide*.

Full configuration decoder Type 1 configuration cycles are used to access the configuration spaces of PCI-to-PCI bridges.

The full configuration decoder behaves like the configuration decoder, and additionally reacts to type 1 configuration cycles that access a subordinate bus. A type 1 configuration cycle is recognized by AD[1:0]=01 (whereas type 0 cycles use AD[1:0]=00).

The transaction is claimed if the bus number accompanying the transaction is equal to or lies between the bus numbers of the primary and the secondary bus. The bus number is taken from AD[23:16].

- The *primary bus* is the first subordinate bus directly connected to the bridge's "downward" interface.
- The *secondary bus* is the subordinate bus with the highest bus number.

Subtractive Decoder The subtractive decoder claims all transactions that are not claimed by another device. This is the behavior of ISA bridges.

CAUTION

Do not use the testcard with subtractive decoder if an ISA bridge resides on the bus because hardware might be destroyed.

Priorities and Parameters

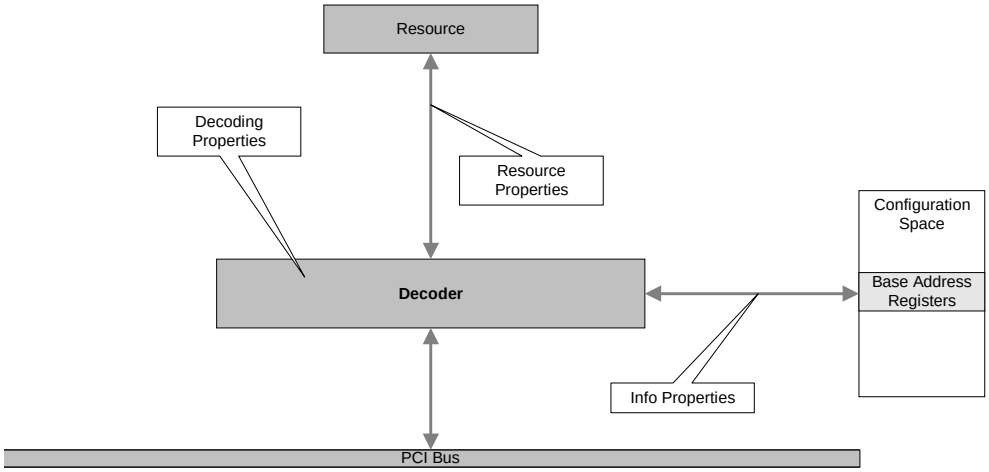
A transaction is allowed to be decoded by one decoder only. Hardware damage would be the result otherwise. However, the decoder of the testcard can be set up with overlapping memory ranges. The decoders of the testcard, therefore, provide priorities. These priorities regulate which decoder claims a transaction that is in the range of several decoders.

The following table shows the target decoder properties to be used with each decoder and the priority of each decoder (1 is the highest priority).

Prio.	Decoder	Properties
1	Fast Configuration Decoder	Configuration Decoder properties and property Speed. Refer to "*Target Decoder Properties*" in the *Agilent E2925B Opt. 300 Exerciser User's Guide*.
2	Fast I/O Decoder (Standard Decoder 1 – 6)	Standard Decoders 1 – 6 properties and property Speed. Refer to "*Target Decoder Properties*" in the *Agilent E2925B Opt. 300 Exerciser User's Guide*.
3	Fast Memory Decoder (Standard Decoder 1 – 6)	
4	Expansion ROM Decoder	Base Address, Size Speed (except fast) Resource, Internal Address, Size
5	Standard Decoder 6	Behavior Base Address, Size, Mask Base Decoder (if "Behavior" = overlay) Commands Dual Address Cycle (DAC) Initialization Device Select (IDSEL) Location Prefetchable Resource, Internal Address, Res. Size Speed
6	Standard Decoder 5	
7	Standard Decoder 4	
8	Standard Decoder 3	
9	Standard Decoder 2	
10	Standard Decoder 1	
11	Configuration Decoder	Base Address, Mask Resource, Internal Address, Size Speed
12	Full Configuration Decoder	Base Address, Mask Initialization Device Select (IDSEL) Bus Numbers Speed (except fast) Resource, Internal Address, Res. Size
13	Subtractive Decoder	Speed (all) Resource, Internal Address, Res. Size Commands Dual Address Cycle (DAC) Initialization Device Select (IDSEL)

Decoder Properties

The target decoder properties determine the decoder behavior in all respects. In the following description the properties are grouped according to their function. The figure below gives an overview:



- **Decoding Properties**

  These properties concern the decoding process itself (basic properties), such as base addresses, decoded commands, decoder behavior, and so forth.

- **Info Properties**

  These properties describe the decoded address range: the type (memory or I/O), location, prefetchability, and so forth.

- **Resource Properties**

  These properties determine the resource connected to the decoder.

Decoding Properties

The following properties determine the decoding behavior of the decoders:

- **Behavior**

  This property defines the behavior of a standard decoder: *normal*, *overlay*, *config*, or *custom*, as described in *"Decoder Behavior(s)" on page 118*.

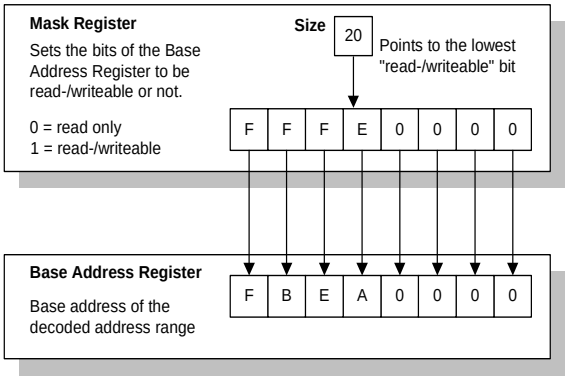
  The setting of the behavior property can disable or limit the other decoding properties. The way it influences which decoding property is shown in the following table:

Decoding Properties	Normal Behavior	Overlay Behavior	Configuration Behavior	Custom Behavior
Base Address	programmable	programmable within Base Decoder Range	programmable (low address only)	programmable
Mask	ignored programmable by "Size"	ignored programmable by "Size"	programmable (low mask only)	programmable
Size	programmable	programmable within Base Decoder Range	ignored (programmable by Mask)	ignored (programmable by Mask)
Base Decoder	ignored	Base Decoder Identifier	ignored	ignored
Location	programmable	ignored (equals Base Decoder)	ignored (derived from base address)	ignored (derived from base address)
Prefetch	programmable	ignored (equals Base Decoder)	ignored (derived from base address)	ignored (derived from base address)
Speed	programmable	ignored (equals Base Decoder)	programmable	programmable
Command	ignored derived from Location	programmable: Subset of Base Decoder Commands	programmable: Config Read/ Config Write	programmable
Dual Address Cycle	ignored derived from Location	ignored (equals Base Decoder)	off	programmable
IDSEL	ignored	ignored (equals Base Decoder)	programmable	programmable

- **Mask, Size, Base Address**

  The address range to be decoded is specified with these properties. They refer to the base address register entry of the configuration space.

  The *base address* property specifies the base address of the range, the size of the address range is specified by either *size* or *mask* (depending on the setting of the “behavior” property).



  – **Mask and Base Address Register**

    The *Mask* property sets the *Mask Register*, which specifies which bits of the *Base Address Register* are “read-only” or which are “read-/writeable”. The “read-/writeable” bits are used by the BIOS to determine the memory size of a PCI device and to enter the base address during configuration.

  – **Size**

    The *Size* property specifies the size of the required memory. This is for convenience and is intended to be used for standard decoders with “normal” behavior.

    Size is an integer. It is the exponent of a value with the base 2 (actual size = 2^{size}). For example: when size is 20, the actual size of the memory is 2^{20} (1 MB). A value of 0 switches the decoder off.

    The memory size will be recalculated internally into a *mask* value: the size value points to a bit in the mask register. This bit and all the higher bits of the mask register are set to 1 (and are therefore “read-/writeable” in the base address register).

    Example:

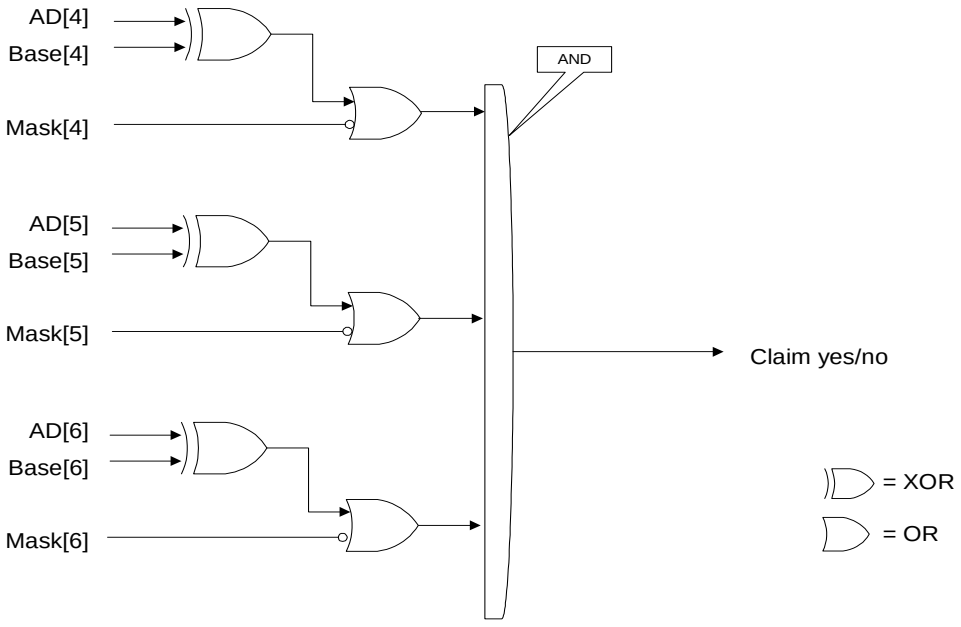
    The base address is `FBEA 0000\h`. To decode all addresses up to `FBEA FFFF\h`, set either size to 20 (decimal) or mask to `FFFE 0000\h`.

CAUTION

Programming overlapping address ranges to different devices may cause hardware damage!

The decoders of the testcard can be set up with overlapping address ranges, however, decoder priorities prevent the testcard from being damaged.

The following figure shows, as an example, the circuits for the bits 4 to 6 of address lines, the base address register setting and the mask register setting, when logically linked for decoding.



When decoding, the decoder compares the settings of address line, base address register and mask as shown in the figure. If the comparisons return “true” for all bits, the transaction will be claimed (if IDSEL and the bus command match as well).

This procedure is basically the same for each type of decoder. However, during configuration cycles, the signals on the address lines and the bits in the mask register do not transfer address information. For information on the meaning of the address lines during configuration cycles, please refer to the PCI Specification.

- **Base Decoder for Overlay Behavior**

  This property points to the base decoder and is used only if the decoder behavior is “overlay”. All parameters from the base decoder are also used for the overlay decoder, however you can specify a subset of commands and/or a smaller address range, and a different resource.

  Refer to *“Decoder Properties” on page 111*.

- **Commands**

  With this property, PCI commands can be excluded from decoding. This makes it possible, for example, to set up a decoder just to decode reads.

  Which commands are available depends on the decoder type and behavior. For example, a configuration decoder with normal behavior will not decode memory commands.

- **Dual Address Cycle (DAC)**

  This property enables 64-bit capability to the decoder. This is only applicable to standard decoders 1, 3 and 5. The referring neighbor decoder is then also used for 64-bit decoding. For example, decoders 1 and 2 decode a complete 64-bit address.

  The subtractive decoder can be set to decode both, 32- and 64-bit addresses.

- **IDSEL (Initialization Device Select)**

  The IDSEL signal controls whether or not a configuration transaction is claimed. IDSEL is set during a configuration cycle by the master (usually the host bridge) to address the target with the configuration space it wants to access.

  Using this property, the full configuration decoder and the subtractive decoder of the testcard can be set up to decode configuration transactions that are not addressed to it. The decoder can be set up to claim

  – all configuration transactions (independent of the IDSEL signal)

  – only configuration transactions coming with IDSEL

  – only configuration transactions coming without IDSEL

- **Bus Numbers for Type 1 Configuration Cycles**

  For type 1 configuration cycles, the primary and secondary bus number is set by these properties. The configuration cycles will be decoded if the bus number of the bus addressed by the transaction is in the range spanned by these bus numbers.

- **Speed**

  This property determines the decode speed (DEVSEL# timing) of the testcard. The following speeds are available:

  – **Fast**

  Memory, I/O and configuration (type 0 only) decoders can decode fast. Maximum speed, however has the following limitations:

  - The *resource* must be the data memory or compare unit.
  - For a *memory decoder*, the decoded address range and size are limited. Behavior must be “normal” or “config”.
  - For an *I/O decoder*, the decoded address range and size are limited. Behavior must be “normal” or “config”.
  - For a *configuration decoder*, IDSEL must be asserted. For decoding, only the bits that represent the function of a multifunctional device are taken into account (AD[10:8]), but not the register within the configuration space of that device’s function (AD[2:7]).
  - No address range checking is performed. The decoder can accept bursts that exceed the upper limit of its address range.

  – **Protected Fast**

  This property forces single cycles (alternating address and data phases) and therefore address range checking during fast decoding, so that the decoder cannot accept bursts and exceed the upper limit of its address range anymore.

  – **Medium and Slow**

  These properties set the testcard’s DEVSEL# timing and the referring entries in the configuration space to medium or slow.

  – **No DEVSEL#**

  The decoder will not assert DEVSEL# to answer a master’s request, although a transfer meets its decode address range.

  This forces the requesting master to abort after a time because no device answers its request. If, however, a subtractive decoding device is connected to the bus, this can still claim the transaction.

Info Properties

The info properties describe the decoded address range. They refer to the entries in the Base Address Registers in the configuration space header of the testcard. They can be read there by the BIOS during configuration cycles. The BIOS must regard the settings when allocating system memory resources for the testcard.

- **Location**

  Available locations are:

  – in 32-bit range

  – in 64-bit range

  – below 1 MB

  – in I/O range

- **Prefetchable**

  This property specifies whether memory is prefetchable and, thus, whether a master can take advantage of optimized access to the memory of the target. The property is not available for the I/O range.

Resource Properties

The following properties describe the testcard's internal resource, which is connected to the decoder.

- **Resource**

  Available resources are:

  – Data Memory or Compare Unit

  – Configuration Space

  – CPU Port

  – Static I/O

  – Expansion ROM

  For an overview of the available resources, refer to *"Data Resources"* in the *Agilent E2925B Opt. 300 Exerciser User's Guide*.

- **Internal Address and Size**

  These properties determine the memory range of a (memory) resource. Using different internal addresses of the testcard allows the specification of partitions of the internal data memory as resources, which then can be used by different decoders.

  The internal address range does not necessarily need to span the same size as the address range that the decoder decodes. The internal memories are cycled through. This allows, for example,

  – specification of a size of 1 to simulate the front end of a FIFO address, as they are often used for I/O

  – acceptance of bursts that are longer than the available memory space

Decoder Behavior(s)

For each Standard Decoder of the testcard, a particular behavior can be programmed. This allows the testcard to be set up for certain types of tests. To transfer data using the testcard, you need the following decoder behavior (which is also the default):

- **Normal Behavior**

  This programs the decoder to behave as specified in the Base Address Registers. It is intended for PCI-compliant memory and I/O decoding.

- **Overlay Behavior**

  This allows connection of different resources to one address range.

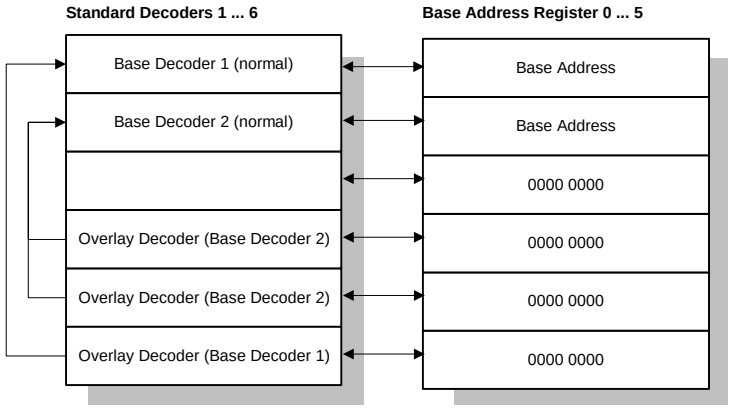
  If, for example, you need different internal resources to store data received from and to be driven onto the PCI bus, you *cannot* simply set up two different decoders—because one PCI address (range) can be decoded by one decoder only. Hardware damage would be the result otherwise. Therefore, the decoders of the testcard provide priorities.

NOTE You can use a command subset as well as an address (range) subset.

Instead, you program a base decoder (the “real” decoder) and a decoder with “overlay behavior”. To the decoder with overlay behavior, you assign a subset of the commands of the base decoder and its own resource. The decoder with overlay behavior will then redirect the data transferred with these commands to/from that resource.

The base address register of a decoder with overlay behavior will always be set to zero. Base address registers following a base address register set to zero in the configuration space header will not be recognized by the BIOS. Therefore, use the lower standard decoders as base decoders and locate the overlay decoders after them.

The base decoder must be set up to “normal” behavior. You can use one base decoder with multiple decoders with “overlay” behavior. The figure below shows an example:



- **Config Behavior**

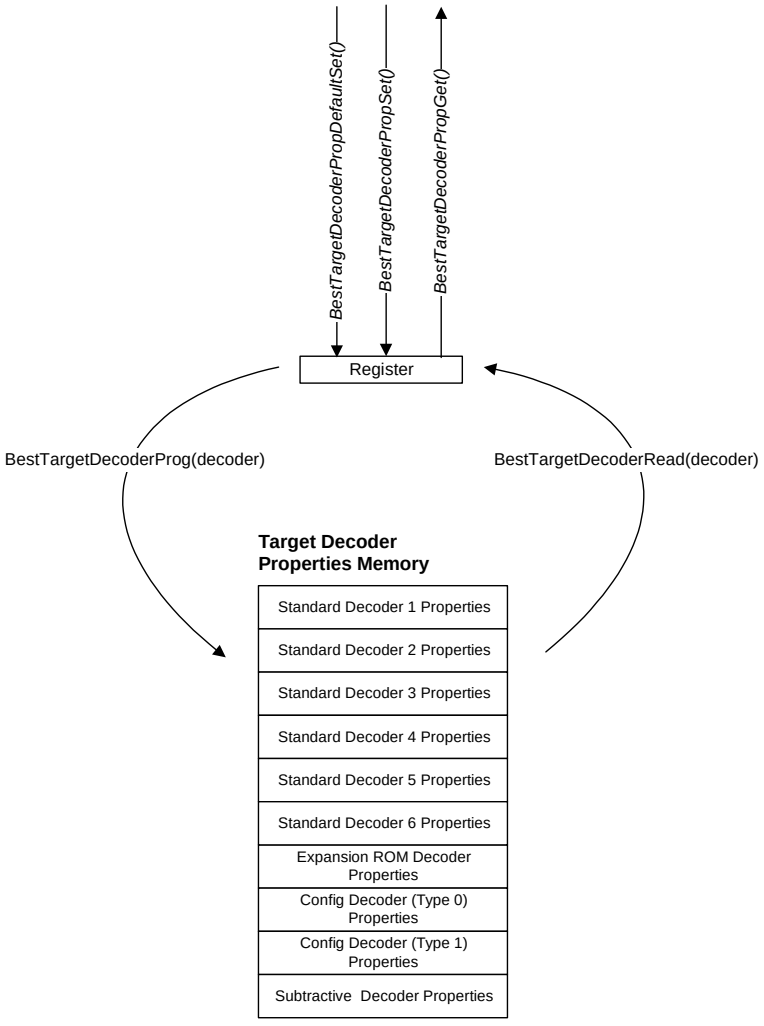
  This programs the decoder to behave like a PCI-compliant configuration decoder (type 0 only). For example, if this behavior is programmed to the six standard decoders, this emulates six devices reacting on configuration cycles from the host bridge.

- **Custom Behavior**

  This behavior allows the programming of non-PCI-compliant decoders. You can set the programmable properties of a decoder according to your testing requirements. Your settings are not checked for PCI compliance.

Functions Overview

The following figure shows the functions used to program the target decoder properties memory.



Programming Steps

Programming Target Decoder Properties requires the following steps:

1. Set the preparation register for a specific decoder to default values. Use *BestTargetDecoderPropDefaultSet*.
2. Change decoder properties in the preparation register as needed. Use *BestTargetDecoderPropSet*.
3. Program the contents of the preparation register for the specific decoder to the memory.

Example

Task

Program the testcard to react to BIOS configuration cycles by requesting a memory area of 16 KByte located in the 32 bit address range.

Proceed as follows:

1 Set up the testcard to decode accesses to this memory using a 32-KByte internal data memory area as follows:

- The data from write accesses gets stored in a lower 16-KByte area.
- The data for read accesses is supplied from the upper 16-KByte area.

To do this, you need to set up two standard decoders to direct read and write transfers to different resources: One with normal and one with overlay behavior.

2 Program the *decoder with overlay behavior* to decode a command subset of its *base decoder*.

Because the decoder with overlay behavior has a higher priority, the base decoder will only decode those transactions that are not claimed by the decoder with overlay behavior.

Set up the Standard Decoder 1

3 Set up the Standard Decoder 1 to be the base decoder with:

- Normal behavior
- 16 KByte decoded address range (= 2^{14}, therefore *size* is 14) in 32-bit space, (non-prefetchable)
- Medium decode speed
- Internal resource 16 KByte data memory, starting with internal address `0x00`

Implementation

```
/* Set up Standard Decoder 1. */
err=BestTargetDecoderPropDefaultSet(handle,
B_DEC_STANDARD_1);C(err);

err=BestTargetDecoderPropSet(handle, B_DEC_BEHAVIOR, NORMAL); C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_SIZE, 14);C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_LOCATION, B_LOC_SPACE32);
C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_PREFETCH, 0);C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_SPEED, MEDIUM);C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_RESOURCE, B_RES_DATA);
C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_RESBASE, 0x0000);C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_RESSIZE, 0x4000);C(err);

/* Program the decoder settings to the memory. */
err=BestTargetDecoderProg(handle, B_DEC_STANDARD_1);C(err);
```

Set up the Standard Decoder 6

The following lines set up the Standard Decoder 6 as a *decoder with overlay behavior*:

- behavior is overlay, base decoder is Standard Decoder 1
- read commands are redirected to upper 16 kByte of internal memory

Implementation

```
err=BestTargetDecoderPropDefaultSet(handle, B_DEC_STANDARD_6);
C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_BEHAVIOR, B_BEH_OVERLAY);
C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_OVERLAY,
B_DEC_STANDARD_1); C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_BUSCMD,
                     \(B_CMDBIT_MEM_READ \
                     |B_CMDBIT_MEM_READLINE \
                     |B_CMDBIT_MEM_READMULTIPLE);C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_RESOURCE, B_RES_DATA);
C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_RESBASE, 0x4000);C(err);
err=BestTargetDecoderPropSet(handle, B_DEC_RESSIZE, 0x4000);C(err);

/* Program the decoder settings to the memory. */
err=BestTargetDecoderProg(handle, B_DEC_STANDARD_6);C(err);
```

Target Attribute Memory Programming

Memory Contents

The target attribute memory contains protocol level information on how decoded transactions should be performed.

That means, it contains all properties of a bus phase during a transfer, such as how many waits are to be inserted into a data phase or whether a parity error should be signaled during an address phase, and so forth. The attributes can be used for data transfers to or from the internal data memory and compare unit. The attribute memory is shared by all decoders.

The settings of this memory do not affect the result of the data transfer. You can repeat transfers with **fixed target decoder** settings but with **varying target attributes,** and then compare the transferred data with data stored in the testcard's internal memory. For example, the target attributes could be varied by a randomizer.

A target attribute memory line covers all types of target attributes:

Address Phase Attributes

The exerciser uses the address phase attributes only during an address phase of a transaction. At other transaction phases, the address phase attributes are ignored. You can find all address phase attributes with their detailed descriptions in *"Address Phase Attributes (Target)"* in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

Data Phase Attributes

The exerciser uses the data phase attributes only during a data phase of a transaction. You can find all address phase attributes with their detailed descriptions in *"Data Phase Attributes (Target)"* in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

Control Attributes

The control attributes are used to determine how the exerciser works through the lines of target attribute memory. You can find all address phase attributes with their detailed descriptions in *"Control Attributes (Target)"* iin the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

The next memory line is selected after each data phase**, regardless of whether or not it was successful**. The address phase attributes of a line are ignored during a data phase.

Memory Design

The target attribute memory can hold up to **256 lines**. It is organized in **64 pages** with **4 lines** each. Page 0 contains default values and is read-only.

NOTE

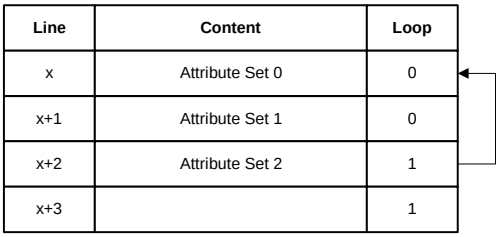
For compatibility with older C-API versions, there is also an 8-page by 32-line memory organization. This is also the default. To take advantage of the enhanced number of pages, change the target attribute page size (using *BestExerciserGenPropSet*) and store this as power-up property so that this setting will be used at every power-up. (This does not affect the principles described in the following.)

Each line represents one bus phase, **regardless of whether it is successfully performed or not** (this is different than the behavior of the master), and contains both address and data attributes.

The “end of page” information is held in a loop bit. The loop bit needs to be set to “0” except in the last memory line. In this way, it is ensured that the exerciser cycles through the target attributes and returns to the beginning.

Building Loops in the Target Attribute Memory

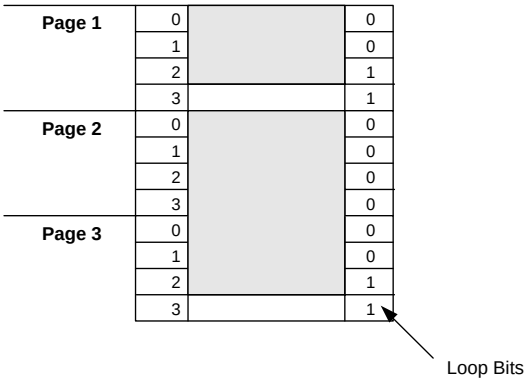
The following figure shows how loops are built in the target attribute memory.



Line	Content	Loop
x	Attribute Set 0	0
x+1	Attribute Set 1	0
x+2	Attribute Set 2	1
x+3		1

Programming Concatenated Pages

The following figure shows how pages with more than 4 lines are programmed in concatenated pages. The figure is also used to explain how the exerciser works through the target attribute memory.



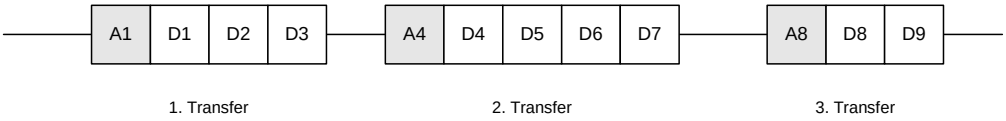
Page 1 contains 3 entries with target attributes. It is referred with value 1. The exerciser works through the lines until it finds the row with a loop bit set to 1, and then restarts the page with the next phase. It continues in this way until the transaction ends.

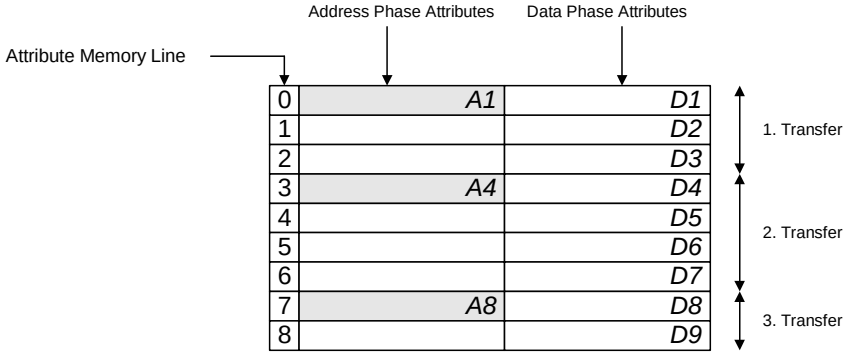
Line 3 in page 1 can be left on its default values. It is out of the loop and will not be executed.

Pages 2 and 3 are concatenated pages and contain 7 lines with target attributes, which are worked through by the exerciser when it refers page 2. A reference to page 3 would result in an error (except for initialization). Line 3 of page 3 can again be left as it is, because it is out of the loop and will not be executed.

Working through the Memory

The following figure shows how the attribute memory is worked through with each bus phase:





The figure shows three transfers, each one consisting of one address phase and several data phases. There is one column for the attributes for address phases, and another column for the attributes for data phases.

The attributes for one transfer must be programmed into consecutive lines. There is one line for each data phase. In each line

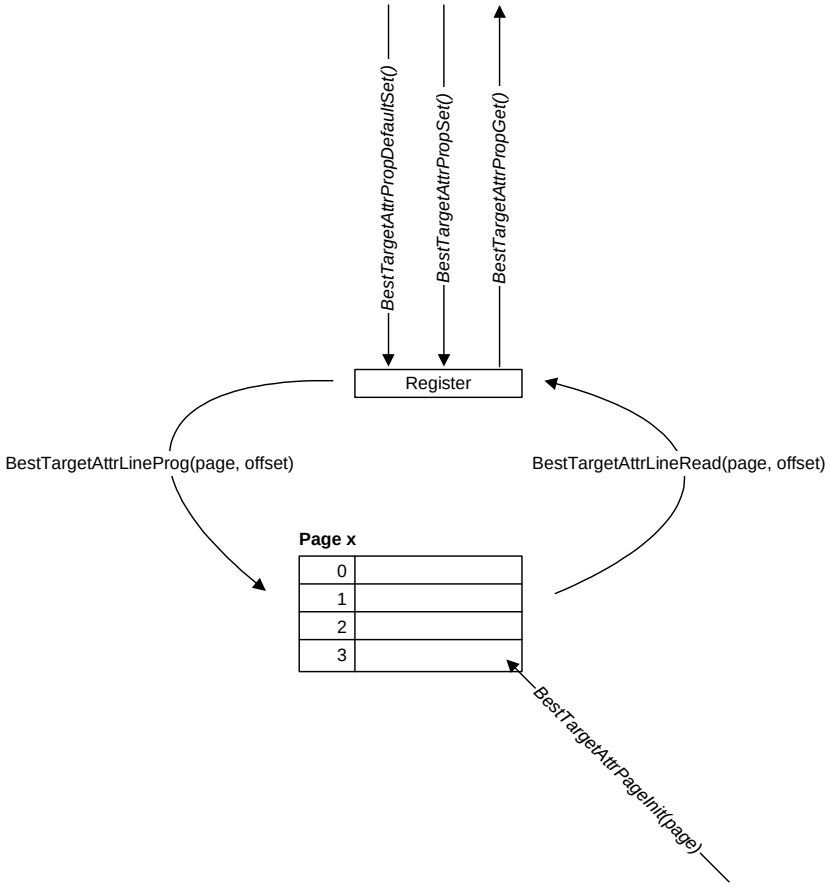
- the first column holds the address phase attributes (the same attributes for each line belonging to one transfer),
- the second column holds the data phase attributes for one data phase (different attributes for different data phases).

Normally only the address phase attributes in the first line for a transfer are used, those in the following lines are ignored. If, however, the master *must* continue with an address phase (for example, after a target retry that occurred within a transaction), the target will use the target address phase attributes from the current line. Otherwise they are ignored.

To achieve a deterministic attribute behavior of the target, each transfer beginning with an address phase must start with the beginning of a target attribute page. See “Run Mode” in *“Programming Generic Target Properties” on page 107*.

Functions Overview

The following figure shows the C functions used when programming target attribute pages.



Programming Steps

Programming target attributes requires the following steps:

1. Initialize a target attribute page. The page will be identified by its number.
   Use *BestTargetAttrPageInit*.
2. Set the preparation register to default values.
   Use *BestTargetAttrPropDefaultSet*.
3. Change attributes in the preparation register as needed.
   Use *BestTargetAttrPropSet*.
4. Program an attribute memory line with the content of the preparation register.
   Use *BestTargetAttrLineProg*.

5 Repeat steps 3 and 4 for each line (address/data phase) you want to program in the attribute page.

For each line within the loop, leave the loop bit at 0, but in the last line of the loop set this bit to 1.

6 Repeat these steps for all the attribute pages you need.

7 Select the page to be used (per session).
Use *BestTargetAttrPageSelect*.

Example

Task Program and select target attribute page 1, define 3 attribute phases with an increasing number of wait states (3, 5, 7) and a disconnect during the third data phase.

Implementation

```
/* Target attribute page 1: set protocol behavior. */
err=BestExerciserGenPropSet(handle, B_EXE_ATTRPAGESIZE, 4); C(err);

/* Initialize the attribute page. */
err=BestTargetAttrPageInit(handle, MyAttrPage); C(err);

/* Set the preparation register to default values */
err=BestTargetAttrPropDefaultSet(handle); C(err);

err=BestTargetAttrPropSet(handle,B_T_WAITS,3); C(err);

err=BestTargetAttrLineProg(handle, MyAttrPage, 0); C(err);

err=BestTargetAttrPropSet(handle,B_T_WAITS, 5); C(err);

err=BestTargetAttrLineProg(handle, MyAttrPage, 1); C(err);

err=BestTargetAttrPropSet(handle,B_T_WAITS, 7); C(err);

err=BestTargetAttrPropSet(handle, B_T_TERM, B_TERM_DISCONNECT);
C(err);

err=BestTargetAttrPropSet(handle,B_T_DOLOOP, 1); C(err);

err=BestTargetAttrLineProg(handle, MyAttrPage, 2); C(err);

err=BestTargetAttrPageSelect(handle, MyAttrPage); C(err);
```

Target Attribute Groups Programming

To achieve more sophisticated randomization opportunities, the target attributes are divided into groups, which can be varied against each other. For this purpose, the C-API provides an own function set controlling the attributes and the loop bit per group.

Attribute Groups

The following table shows which attribute is assigned to which group:

Group	Address Phase Attributes
TA0	64-Bit Acknowledge System Error Signalling

Group	Data Phase Attributes
TD0	Waits
TD1	Termination, Parity and System Errors
TD2	Marker

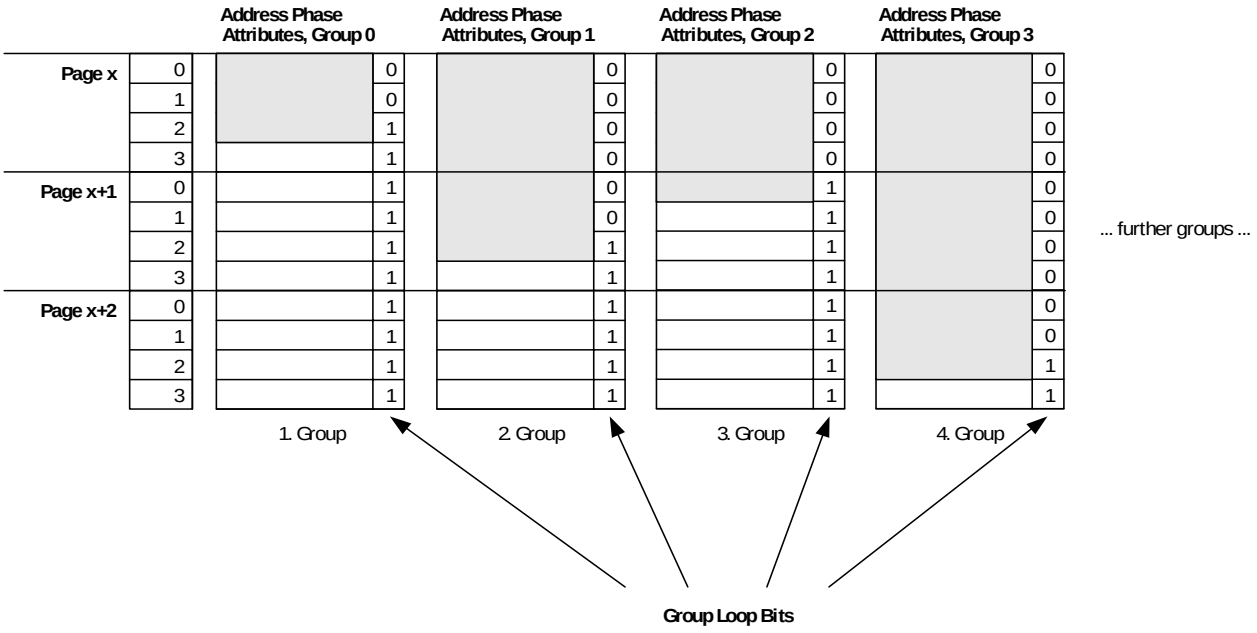
Group	Control Attributes
TC	Repeat

This assignment is fixed and cannot be programmed or otherwise changed.

Composing a Target Attribute Page

A target attribute page can be composed of particular group pages of different sizes.

The following figure shows an example:



The figure shows four groups of page x:

- The first group is address phase attribute group 0.
- The second group is address phase attribute group 1.
- The third group is address phase attribute group 2.
- The fourth group is address phase attribute group 3.

In the figure, the remaining groups are skipped for clearness.

The default page size of 4 lines is exceeded by all groups except the first one. The fourth group exceeds 8 lines. Therefore, pages “x+1” and “x+2” are concatenated and also part of page x.

The fourth group provides the most lines—11—and thus determines the overall page size of page x. The next available page is page x+3.

When the exerciser calls page x, the attributes of the groups are cycled through *independently* per transaction phase. This means the following:

- The first three phases use lines 0 to 2 of page x over all groups.
- Phase four uses line 3 of the second, third and fourth group—but line 0 of the first group: this group restarts at its page begin.

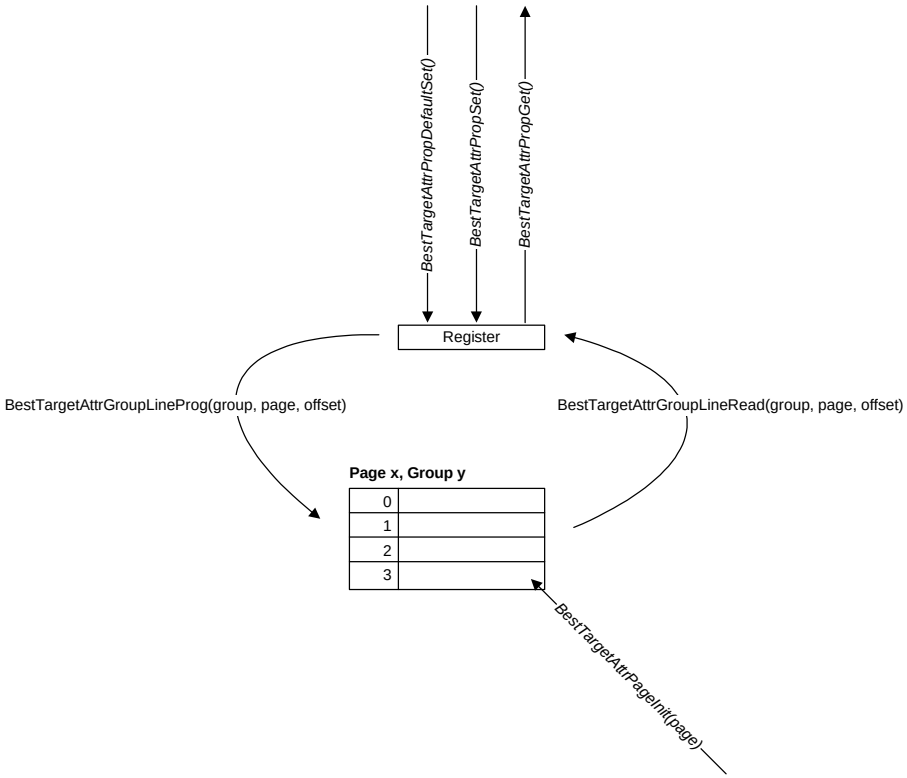
- The third group restarts in the sixth phase. At this moment, the first group will have cycled through the second time and is “standing” on line 2.
- The second group restarts in phase eight. At this moment, the first group is in line 1 and the third group is in line 2.
- The fourth group restarts in phase 12. At this moment, the first group is in line 2, the second is in line 4, the third is in line 1.

NOTE The group page sizes must always be prime numbers and must always differ to vary the attributes of the groups against each other.

Function Overview

NOTE Within a page, these functions are not allowed to be mixed with the non-group functions.

The following figure shows the functions used to program target attribute groups to the memory.



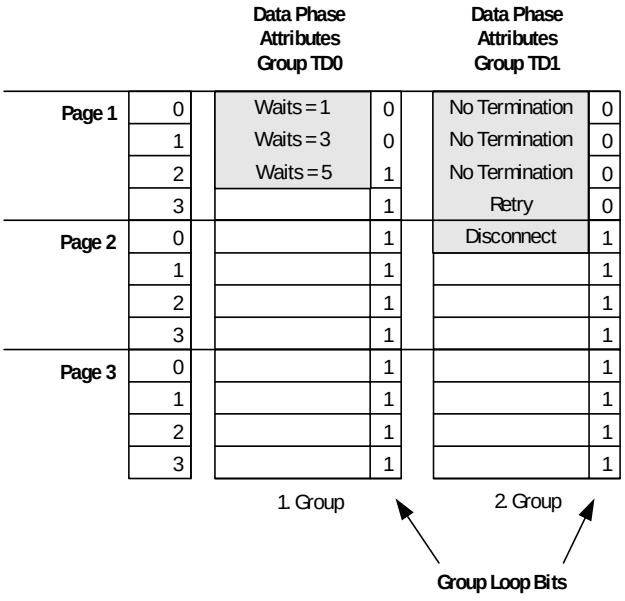
Programming Steps Programming target attributes requires the following steps:

1. Initialize a target attribute page. The page will be identified by its number.
   Use *BestTargetAttrPageInit*.
2. Set the preparation register to default values.
   Use *BestTargetAttrPropDefaultSet*.
3. Change attributes in the preparation register as needed for one group.
   Use *BestTargetAttrPropSet*.
4. Program an attribute memory group line with the content of the preparation register.
   Use *BestTargetAttrGroupLineProg*.
5. Repeat steps 3 and 4 for all groups you want to program in one line.
6. Repeat steps 3, 4 and 5 for each line (address/data phase) you want to program in the attribute page.

   For each line within the group, leave the loop bit at 0, but in the last line of the group set this bit to 1. Use attribute B_T_LOOP.
7. Repeat these steps for all the attribute pages you need.
8. Select the page to be used (per session).
   Use *BestTargetAttrPageSelect*.

Examples

Task Program the following target attribute group pages.



Implementation

```
pg_num=1;

offset=0;

/* Initialize attribute page 1. */
err = BestTargetAttrPageInit(handle, 0x01); C(err);

/* Set the preparation register to default values. */
err = BestTargetAttrPropDefaultSet(handle); C(err);

/* Set the attributes for line 0 for group TD0 (target data0) in
the preparation register. */
err = BestTargetAttrPropSet(handle, B_T_WAITS, 0); C(err);

/* Program the attribute group TD0 to set waits in line 0 of the
attribute memory. */
err = BestTargetAttrGroupLineProg(handle, B_TATTR_GRP_TD0, pg_num,
offset); C(err);

/* Set the attributes for line 0 for group TD1 (target data1) in
the preparation register. */
err = BestTargetAttrPropSet(handle, B_T_DOLOOP, 0); C(err);
err = BestTargetAttrPropSet(handle, B_T_TERM, B_TERM_NOTERM);

/* Program the attribute group TD1 to set terminations in line 0 of
the attribute memory. */
err = BestTargetAttrGroupLineProg(handle, B_TATTR_GRP_TD1, pg_num,
offset); C(err);

offset++;

/* Set the attributes for line 1 for group TD0 in the preparation
register. */
err = BestTargetAttrPropSet(handle, B_T_WAITS, 3); C(err);

/* Program the attribute group TD0 to set waits in line 1 of the
attribute memory. */

err = BestTargetAttrGroupLineProg(handle, B_TATTR_GRP_TD0, pg_num,
offset); C(err);

/* Set the attributes for line 1 for group TD1 in the preparation
register. */

err = BestTargetAttrPropSet(handle, B_T_TERM, B_TERM_NOTERM);

// Program the attribute group TD1 to set terminations in line 1 of
the attribute memory. */

err = BestTargetAttrGroupLineProg(handle, B_TATTR_GRP_TD1, pg_num,
offset); C(err);

offset++;
```

```
/* Set the attributes for line 2 for group TD0 in the preparation
register. (Set the loop bit for group TD0). */
err = BestTargetAttrPropSet(handle, B_T_WAITS, 5); C(err);
err = BestTargetAttrPropSet(handle, B_T_DOLOOP, 1); C(err);

/* Program the attribute group TD0 to set waits in line 2 of the
attribute memory. */
err = BestTargetAttrGroupLineProg(handle, B_TATTR_GRP_TD0, pg_num,
offset); C(err);

/* Set the attributes for line 2 for group TD1 in the preparation
register. */

err = BestTargetAttrPropSet(handle, B_T_DOLOOP, 0); C(err);
err = BestTargetAttrPropSet(handle, B_T_TERM, B_TERM_NOTERM); C(err);

/* Program the attribute group TD1 to set terminations for line 2
of the attribute memory. */
err = BestTargetAttrGroupLineProg(handle, B_TATTR_GRP_TD1, pg_num,
offset); C(err);


offset++;

/* Set the attributes for line 3 for group TD1 in the preparation
register. */
err = BestTargetAttrPropSet(handle, B_T_DOLOOP, 0); C(err);
err = BestTargetAttrPropSet(handle, B_T_TERM, B_TERM_RETRY);

/* Program the attribute group TD1 to set terminations in the
attribute memory. */
err = BestTargetAttrGroupLineProg(handle, B_TATTR_GRP_TD1, pg_num,
offset); C(err);


offset ++;

/* Set the attributes for line 2 for group TD0 in the preparation
register. (Set the loop bit for group TD1). */
err = BestTargetAttrPropSet(handle, B_T_DOLOOP, 1); C(err);
err = BestTargetAttrPropSet(handle, B_T_TERM, B_TERM_DISCONNECT);

// Program the attribute group TD1 to set terminations in the
attribute memory. */
err = BestTargetAttrGroupLineProg(handle, B_TATTR_GRP_TD1, pg_num,
offset); C(err);


/* Select this attribute page to be used. */

err = BestTargetAttrPageSelect(handle, 0x01); C(err);
```

Target Run

After adding initialization and control port functions, compile the program and run it on the host. Then reboot the system under test. The BIOS will allocate a base address to Standard Decoder 1 and enable it. After the BIOS configuration phase has been completed, the testcard is ready to run as specified.

You can now start a program that accesses the programmed memory space.

NOTE To run the same decoder set up in a system without BIOS configuration, you would have to specify a base address for Standard Decoder 1 (for example `0xFCE00000`) and you would have to enable it.

To specify the base address, the following line must be added to the lines setting Standard Decoder 1 properties:

```
err=BestTargetPropSet(handle, B_DEC_BASEADDR, 0xFCE00000);C(err);
```

To enable it, add the following lines before storing the settings as power-up defaults.

```
err=BestTargetGenPropSet(handle, B_TGEN_MEMIOSPACE, 1);C(err);
```

It is not necessary to store the settings as power up defaults, unless the system under test is driven without BIOS and will be switched off before the test is run. Therefore, functions *BestPowerUpPropSet* and *BestAllPropStore* could be skipped.

Configuration Space Header Programming

The configuration space header of the testcard enables the testcard to behave like a standard PCI device with a real configuration space. The bits in the configuration space header of the testcard can be set to a certain value and then masked to be read only, read/write and read/clear for configuration access. The setting of the configuration space header can be stored as power-up defaults.

Configuration Registers

The "default value" and "default mask" column in the tables below always shows the factory default. These settings can be restored at any time.

For details of the registers refer to the PCI specification.

Register	Default Mask	Default Value	Notes
Vendor ID	0000	103C	
Device ID	0000	2926	
Command	F900	0280	See *"Command Register" on page 137*. Write accesses are handled in software by the on-board CPU. Read accesses are handled directly in hardware, that is transactions are not delayed.
Status	01D7	0000	See *"Status Register" on page 139*.
Revision ID	0000	0000	
Class Code	0000	0000	
Cache Line Size	FF	00	Any value is accepted. However, the testcard can decide to replace it by a zero (for example, to up the register to be PCI-compliant). The **master** needs this register to be set to a value other than zero to generate MWI cycles with cacheline wrap mode. The **analyzer** uses this register to determine the system's cacheline size. The target does not use this information.
Latency Timer	FF	00	Used by the master.
Header Type	00	00	
BIST	00	00	
Base Address Register 0... 5	Each bit can be programmed to be writeable. Used by the target decoders. See *"Programming the Exerciser as a Target Device" on page 105*.		
	FFFF F000	0000 0008	**BAR 0**: Refers to decoder 1, size is preset to 12.
	FFFF FFF0	0000 0001	**BAR 1**: Refers to decoder 2, size is preset to 4, decoder is preset to decode I/O cycles.
	0000 0000	0000 0000	**BAR 2 ... 5**: Switched off.

Register	Default Mask	Default Value	Notes
Cardbus CIS Pointer	0000 0000	0000 0000	
Subsys. Vendor ID	0000	0000	
Subsystem ID	0000	0000	
Exp. ROM BAR	0000 0000	0000 0000	Bit 0 enables expansion ROM, the remaining bits specify its size. See *"Expansion ROM Programming" on page 141* and *"Programming the Exerciser as a Target Device" on page 105*.
Reserved 0	0000 0000	0000 0000	No function/Capability Pointer.
Reserved 1	0000 0000	0000 0000	No function.
Interrupt Line	FF	00	
Interrupt Pin	00	01	
Min_Gnt	00	00	
Max_Lat	00	00	

Command Register

The following table describes the bits of the configuration space header command register. The "Mask" and "Value" and columns contain factory defaults.

Bits 10 ... 15 are unused. For details, refer to the PCI Specification.

Bit	Mask	Value	Meaning
0	0	0	I/O Space Control. Returns, enables, and disables the capability of the testcard to respond to I/O cycles. Used by the exerciser (target). It takes 4 clocks until the decoders are enabled or disabled.
1	0	0	Memory Space Control. Returns, enables, and disables the capability of the testcard to respond to memory cycles. Used by the exerciser (target). It takes 4 clocks until the decoders are enabled or disabled.
2	0	0	Bus Master Control. Returns, enables, and disables the capability of the testcard to be bus master. Used by the exerciser (master).
3	0	0	Special Cycle Control. Returns, enables, and disables the capability of the testcard to monitor special cycles. It takes 4 clock cycles until the decoders are enabled or disabled.
4	0	0	Memory Write and Invalidate Control. Returns, enables, and disables the capability of the testcard to generate MWI cycles. If disabled, normal memory write cycles are used instead Used by the exerciser (master).

Bit	Mask	Value	Meaning
5	0	0	VGA Palette Snoop Control. No functionality. The testcard can pretend to support “VGA palette snooping” only.
6	0	0	Parity Error Response. Returns, enables, and disables the capability of the testcard to report parity errors, that is: report in status register bit 8, or ignore. This information is used when PERR# or SERR# should be asserted during the address phase due to master or target attribute settings. The testcard does not assert PERR# or SERR# when “real” parity errors occur but only because of attribute settings.
7	0	1	Wait Cycle Control (Address/Data Stepping). No functionality. Whether stepping is actually done depends on the master or target attributes only.
8	1	0	System Error Control. Returns, enables, and disables the capability of the testcard to assert SERR#.
9	0	1	Fast Back-to-Back Control. Returns, enables, and disables the capability of the testcard to perform Fast Back-to-Back cycles. The setting of this bit does not influence the master statemachine.

Status Register The table below describes the bits of the configuration space header status register.

NOTE This is not the status register of the testcard. The “Mask” and “Value” and columns contain factory defaults.

Bits 0 ... 4 are unused. For details refer to the PCI Specification.

Bit	Mask	Value	Meaning
5	0	0	66 MHz Capable Status.
6	1	0	User Definable Features (UDF) Status.
7	1	0	Fast Back-to-Back Status. The target exerciser is not capable of accepting Fast Back-to-Back cycles if the previous transaction was a transaction to a different target.
8	1	0	Data Parity Status. Returns whether PERR# has been signaled.
10:9	00	00	Device Select Timing Status. This setting is independent of the decode speed actually used (see *“Decoder Properties” on page 111*).
11	0	0	Signaled Target Abort Status. Returns whether a target abort has been signaled.
12	0	0	Received Target Abort Status. Returns whether a target abort has been received.
13	0	0	Received Master Abort Status. Returns whether a master abort has been received. Exception: master aborts are not signaled after special cycles.
14			Signaled System Error Status. Returns whether a system error actually used signaled. The bit is set if the exerciser asserts SERR#.
15			Detected Parity Error Status. Returns whether a parity error actually used detected, regardless of whether parity error signalling is disabled. The bit is set if the exerciser asserts PERR# except during address phases.

Functions Overview

The Agilent E2925B testcard's programming interface provides functions for programming the configuration space header. The available functions and their usage is shown by describing the programming steps.

Programming Steps

Programming the configuration space header involves the following steps:

1. Write a value into the configuration space header register.

   Use *BestConfRegSet*.

2. To define registers to be read-only for accesses from other masters, set the mask.

   Use *BestConfRegMaskSet*.

3. Set the power-up property.

   Use *BestPowerupPropSet*.

4. Store the current settings as user defaults for power-up.

   Use *BestAllPropStore*.

Example

Task

Set the "DeviceID" and "VendorID" register in the testcard's configuration space header. These are two neighbored 16-bit registers and are, in the example, accessed by a 32-bit command that programs both registers in one go.

The value they are programmed to is `0x2926103C`, which programs "DeviceID" to `2926` and "VendorID" to `103C`. Mask the registers to be read-only for accesses from other masters.

Implementation

```
/* Set the "DeviceID" and the "VendorID" registers of the testcard.*/
err=BestConfRegSet(handle, 0x00, 0x2926103C); C(err);

/* Set all bits of the registers to read-only. */
err=BestConfRegMaskSet(handle, 0x00, 0x00000000); C(err);

/* Read/write bits will have their factory default values at
powerup */
err = BestPowerupPropSet(handle, B_PU_CONFRESTORE, 0); C(err);

/* Store the current settings of all properties as user defaults
for power-up */
err = BestAllPropStore(handle); C(err)
```

Expansion ROM Programming

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

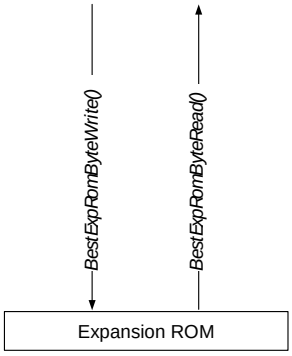
The expansion ROM of the testcard features “code-in-place execution” (XIP), that is without shadowing the expansion ROM content into system memory for execution. This is beyond PCI specification, but can be used in a system in which system memory does not yet work.

The expansion ROM of the testcard is accessible

- by means of C-API functions to fill and read the expansion ROM contents
- through a memory range defined in the “expansion ROM base address register” in the testcard’s configuration space

Functions Overview

The following figure shows the functions used to program the expansion ROM.



The expansion ROM is not programmed using a preparation register. The expansion ROM is simply programmed by …`Write()` and …`Read()` functions, so that no further description is needed.

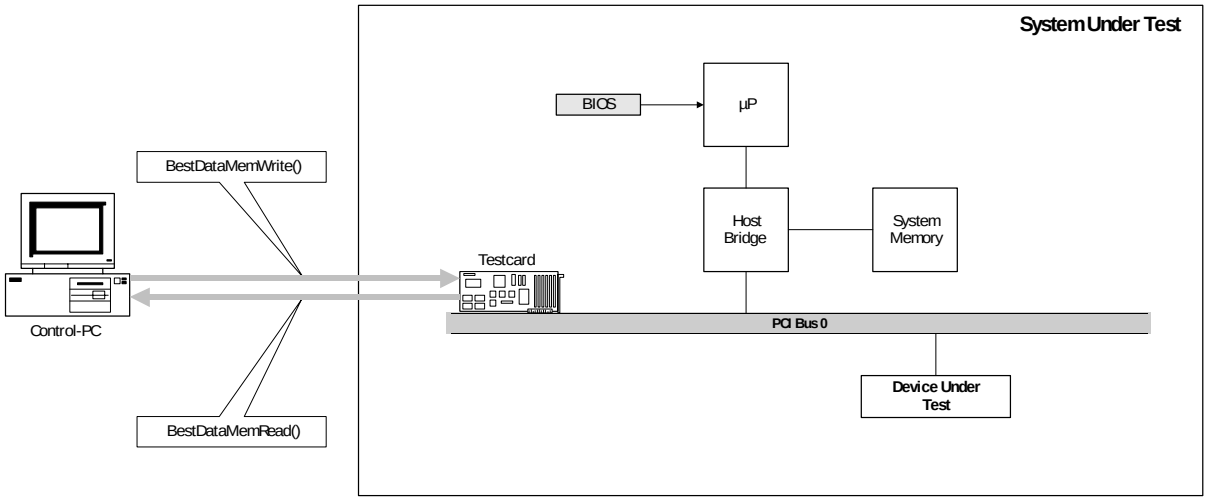
Data Memory and Compare Unit Programming

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.

For detailed information about organization and using of the data memory and the data compare unit, refer to the *Agilent E2925B Opt. 300 PCI Exerciser User's Guide*.

Functions Overview

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



Programming Steps

Programming the data memory and compare unit requires the following steps:

1 Initialize the internal data memory of the testcard, by filling it completely with zeros.

Use *BestDataMemInit*.

2 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 *BestDataMemWrite*.

– To read data from the testcard, the same method is used in reverse.

Use *BestDataMemRead*.

Example

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

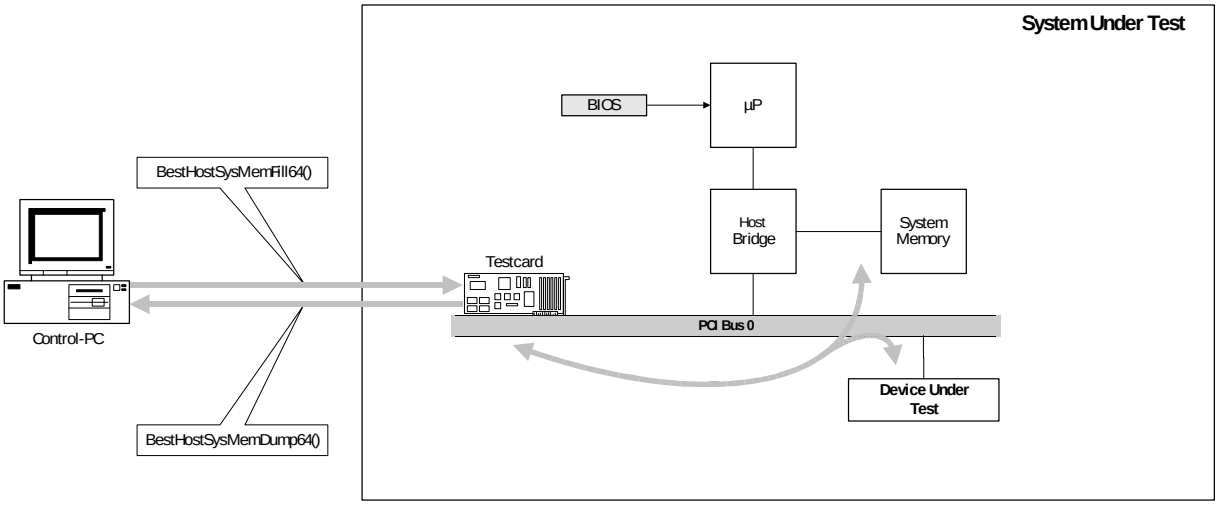
```
err=BestDataMemRead(handle, 0x0000, 0x8000, buffer);C(err);
```

Host Access Programming

The testcard provides *host access functions* that apply **master** block transfers and data memory functions to transfer data between the control PC and the system under test. They are a convenient way for you to transfer data to and from the system under test without having to program master block transfer properties.

Functions Overview

The following figure shows the functions used for host access.



The testcard is plugged into a system under test and connected to a control PC.

Programming Steps

Programming host access requires the following steps:

1 Prepare the transfer.

- Prepare the block transfers used to transfer data from the testcard's data memory to the system under test to allocate the buffer used for the transfers in the data memory of the testcard. (This buffer always resides in the uppermost range of the memory.)
- Define the direction of the master block transfers (read/write).

Use *BestHostSysMemAccessPrepare*.

2 Perform the transfer.

– To transfer data from the control PC to the system under test, use *BestHostSysMemFill64*.

This function transfer data into a buffer in the testcard's data memory. A master block transfer writes the data into the device under test.

– To fetch data from the system under test to the control PC in the opposite direction, use *BestHostSysMemDump64*.

NOTE The functions always use default attributes (attribute page 0) and change generic master properties. See *"Programming the Exerciser as a Master Device" on page 80*.

– For **bytewise** memory transfers, I/O transfers and configuration accesses, use the following functions to set and read PCI registers of devices in the system under test:

Use *BestHostPCIRegGet* and *BestHostPCIRegSet*.

Example

Task Read data from the system under test.

Implementation

```
/* Reserve the uppermost 8 KBytes of the memory for host access and
specify a memory read bus command */

err=BestHostSysMemAccessPrepare(handle, B_CMD_MEM_READ, 8192);
C(err);

/* Read 32 KBytes from a device with PCI address 0xB8000000 to the
memory of the host (memory address is specified by buffer)*/

err=BestHostSysMemDump64(handle, 0, 0xB8000000, 0x0800, buffer);
C(err);
```

Interrupt Programming

The testcard can generate any PCI interrupt INTA# ... INTD#. You can use C functions to:

- Set the interrupt line.

  Use *BestInterruptGenerate*.

- Check the interrupt status in the interrupt status register.

  Use *BestConfRegGet*.

- Clear the interrupt.

  Use *BestStatusRegClear* or *BestConfRegSet*.

These functions can, for example, be used when developing interrupt drivers for a PCI device.

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 **54\h**. The bits of interrupt status register are explained in the table below.

Bit	Oper.	Value	Meaning
0	Interrupt A pending.		
	R	0	No interrupt pending.
		1	An interrupt has been generated by the testcard and waits to be serviced.
	W	x	Ignored.
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.)		
7::4	Reserved		

Example

Task Generate a PCI interrupt and clear all interrupts.

Implementation

```
/* Generate PCI interrupt A */
err=BestInterruptGenerate( handle, B_INTA );C(err);

/* Clear all interrupts */
err=BestConfRegSet( handle, 54\h, 0); B_CHECKERR(err); ;
```

Built-In Test Programming

The built-in test functions set up and perform prepared tests, applying multiple features of the testcard. They simplify programming if the prepared tests meet the testing requirements. The following built-in tests are available:

- Protocol Error Detect

  This test captures protocol errors using the analyzer of the testcard. Refer to *"Protocol Observer Programming" on page 47*.

- Traffic Make

  In order to consume PCI bus bandwidth, this test drives PCI traffic onto the PCI bus. The master of the testcard writes data to the testcard's target.

- Write-Read and Write-Read Compare

  This test accesses another PCI device's memory or the memory of the system under test. Data blocks are written and read via the PCI bus. The read data can be compared with the previously written data.

- Block Move

  This test transfers a data block from one PCI address to another.

- Read

  This test reads data from PCI addresses.

For each test, a set of programmable properties can be used to adapt the test to certain testing requirements. For example, you may vary the protocol stress, block lengths, data pattern, and so forth.

The test results are stored in a waveform file and a report file. The waveform file can be viewed with the listers of the graphical user interface. The report file can be viewed with a normal text editor.

Functions Overview

The Agilent E2925B testcard's programming interface provides functions for set up built-in tests. The available functions and their usage is shown by describing the programming steps.

Programming Steps

Programming the built-in test function, for example a "write-read" test, requires the following steps:

1. Specify new test properties.

   Use *BestTestPropSet*.

2. Specify the master generic properties.

   Use *BestMasterGenPropSet*.

3. Initiate the test run.

   Use *BestTestRun*.

   The test runs until one of the following occurs.

   – A data compare error occurs.

   – The test is stopped externally, for example, by clicking on the *Stop* button in the graphical user interface.

   To stop the test externally, you could, for example, build a loop that continually requests status register contents until one of the register bits indicates that a particular event has taken place. Then the master can be stopped. Refer to "*BestMasterStop*" and "*BestStatusRegGet*" in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

4. Upload the test result.

   Use *BestTestResultDump*.

Example

Task Set up an infinitely running "write-read" test with automatic comparison of the written and read data. Set up the test to write 160 bytes to bus address `b8000\h`, to read them back and compare them with the written data.

Implementation

```
/ * Stop the master and specify the test properties. */
err=BestMasterStop(handle); C(err);

err=BestTestPropDefaultSet(handle); C(err);
err=BestTestPropSet(handle, B_TST_SOURCEADDR, B8000\h); C(err);
err=BestTestPropSet(handle, B_TST_NOFBYTES, 160); C(err);
err=BestTestPropSet(handle, B_TST_DATAPATTERN, B_DATAPATTERN_TOGGLE);
C(err);
err=BestTestPropSet(handle, B_TST_COMPARE, 1); C(err);

/* Set the master to run infinitely in the example, otherwise, the
test will only be performed once. */
err=BestMasterGenPropSet(handle, \
                         B_MGEN_REPEATMODE, \
                         B_REPEATMODE_INFINITE); C(err);

/ * Initiating the test run: The type of built-in test is specified
with the function starting the test: */
err=BestTestRun(handle, B_TSTCMD_WRITEREAD); C(err);

/ * Upload the test results: When the test run has finished the
test results can be written to the files "TESTOUT.WFM" and
"TESTOUT.RPT": */

err=BestTestResultDump(handle, "c:\temp\TESTOUT"); C(err);
```

Just specify the file name, the file suffixes are attached by the function. To view the waveform file, refer to the *Agilent E2925B PCI Analyzer User's Guide*. To view the report file use a text editor.

Programming the Interfaces

The Agilent E2925B testcard provides application interfaces for exchanging information between the testcard and the test environment during the run time of the test application.

The following sections give information about programming these interfaces:

- *"CPU Port Programming" on page 152* shows how the CPU port can be programmed to connect up to two external devices that can supply or pick up data and addresses on separated buses (such as a CPU).
- *"Static I/O Port Programming" on page 160* shows how to connect an external device that can supply and pick up data.
- *"Trigger I/O Sequencer Programming" on page 163* shows how to use the trigger input and output lines.
- *"LED Controlling and Display Functions Overview" on page 169* shows how to write data to the LED display.
- *"Mailbox Programming" on page 171* shows how data can be exchanged between applications running on the control PC and the system under test.
- *"Power Management Event Programming" on page 175* shows how to control power management events on the system under test.

CPU Port Programming

The CPU port of the Agilent E2925B testcard allows the control or initialization of a device with an (Intel-compatible) interface bus. The bus features separate data and address lines. The CPU port includes the following:

- 16-bit address bus (64 Kbyte address range)
- 16-bit data bus
- 2 devices that can be connected directly to ready-to-use select lines
- 3.3 V CMOS outputs driven by 74LVT (can also drive 5 V TTL inputs)
- 3.3 V CMOS inputs connected to 74LVT (can also pick up 5 V TTL outputs)
- automatic timing mode for default timing (no additional RDY# signal necessary)

The outputs of the CPU port can withstand a short.

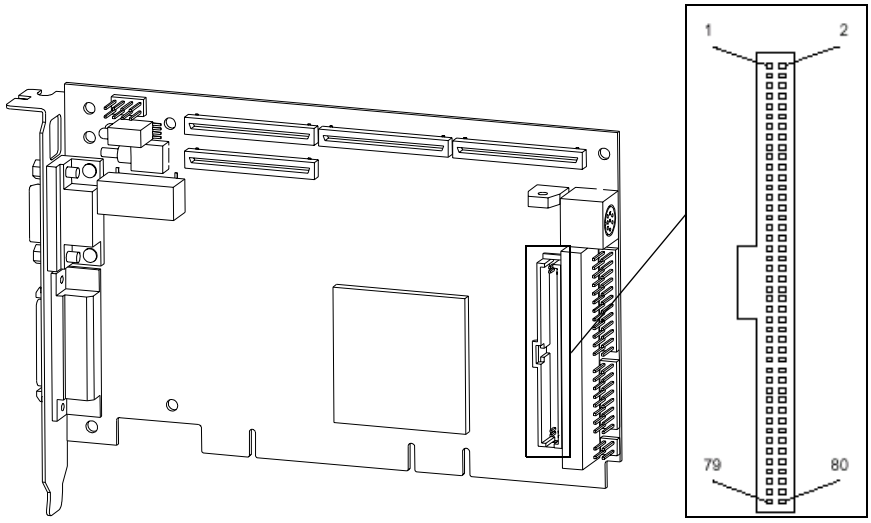
NOTE During transfer, the CPU port is always the master, never a target.

CPU Port Connector

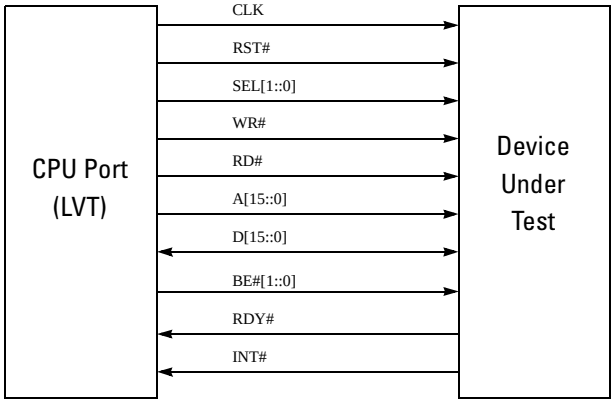
The connector on the testcard is

- Series AMPMODU System 50
- Manufacturer AMP
- Part number 104549-9 (80 positions)

Pin Configuration The figure below shows the connector and the pin configuration of the CPU port of the Agilent E2925B testcard:



Signals of the CPU Port The figure below shows the signals of the CPU port.

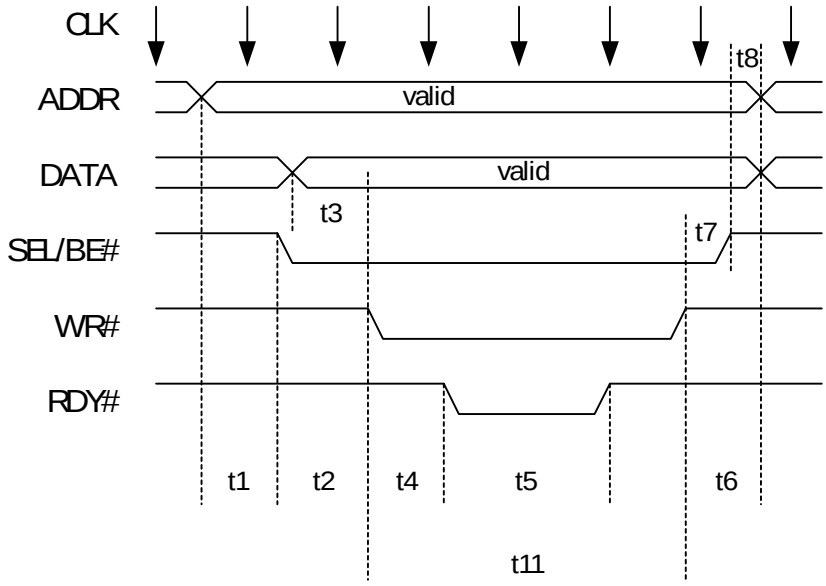


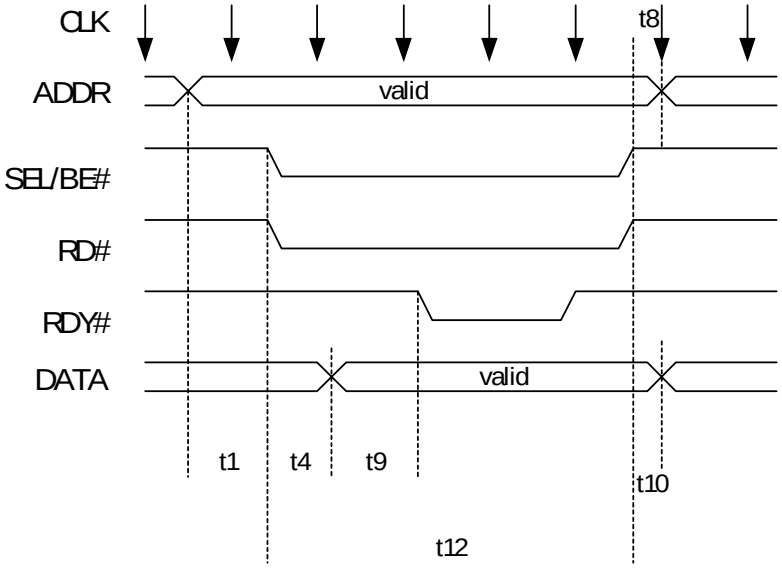
The table shows the direction and the meaning of the signals.

Signal	Dir	Meaning
CLK	Out	16-MHz clock related to the following signals. This signal is helpful but not required.
RST#	Out	Active low reset output.
SEL#[1::0]	Out	Active low chip select signals. Two devices can directly be connected without additional decoding logic.
WR#	Out	Active low write enable signal.
RD#	Out	Active low read enable signal.
A[15::0]	Out	Address bus, A[15] is MSB.
D[15::0]	In / Out	Data bus driven, D[15] is MSB. Output on writes, input on reads.
BE#[1::0]	Out	Active low byte enable signals. On writes, they indicate which byte lines carry meaningful data. On reads, they indicate which byte lines will be used by the testcard.
RDY#	In / -	Active low byte ready signal driven by the device connected to the CPU port. On writes, it indicates that data has been transferred. On reads, it indicates that it has provided the data on the DATA[] bus. In automatic timing mode, this signal does not need to be generated.
INT#	In	Active low, level-sensitive interrupt input.

Timing Diagrams

The figures and the table below contain the timing diagrams for a write and a read and the timing specification for the CPU port signals.





The following table shows the timing specification.

Parameter	Minimum	Notes
t_1	60 ns	
t_2	60 ns	
t_3	50 ns	
t_4	0 ns	N/A in automatic timing mode.
t_5	130 ns	
t_6	60 ns	
t_7	60 ns	
t_8	5 ns	
t_9	0 ns	N/A in automatic timing mode.
t_{10}	0 ns	
t_{11}	160 ns	In automatic timing mode only.
t_{12}	160 ns	

The following table shows the CPU port pin configuration.

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
1	WR#	2	GND	41	A13	42	A14
3	INT#	4	GND	43	A15	44	GND
5	RD#	6	GND	45	D0	46	GND
7	RST#	8	GND	47	D1	48	GND
9	SEL#0	10	GND	49	D2	50	GND
11	SEL#1	12	GND	51	D3	52	GND
13	BE#0	14	GND	53	D4	54	GND
15	RDY#	16	GND	55	D5	56	GND
17	A0	18	GND	57	D6	58	GND
19	A1	20	GND	59	D7	60	GND
21	A2	22	GND	61	BE#1	62	GND
23	A3	24	GND	63	D8	64	GND
25	A4	26	GND	65	D9	66	GND
27	A5	28	GND	67	D10	68	GND
29	A6	30	GND	69	D11	70	GND
31	A7	32	GND	71	D12	72	GND
33	A8	34	GND	73	D13	74	GND
35	A9	36	GND	75	D14	76	GND
37	A10	38	A12	77	D15	78	GND
39	A11	40	GND	79	CLK	80	GND

Connecting the CPU Port to a Decoder

The CPU port is an internal resource that can be connected to a decoder so that the testcard's target can be used for direct transfers between CPU port and PCI bus. Do this according to the following rules:

- Set the decoder properties as follows:
  - "Resource" to "CPU port"
  - "Resource size" to 16
  - "Base address" to 0
- Transfer words and/or dwords only.

The testcard uses the CPU port as it is determined by the CPU port properties (port mode, ready-signal, protocol).

If a decoder and software use the CPU port simultaneously, this can result in conflicts. It is not recommended. Note that locking the CPU port locks the CPU port property settings but not the transfers.

For more information on decoders, refer to *"Programming the Target Decoder Properties Memory" on page 109*.

Functions Overview

Programming Steps

Programming the CPU port requires the following steps:

1 To enable the CPU port, set the CPU port mode to "master".

Use *BestCPUportPropSet*.

2 Optionally, you can wait for an interrupt to occur. For this purpose, read the interrupt status of the CPU port until an interrupt is recognized.

Use *BestCPUIntrStatusGet*.

After the interrupt occurred, clear the interrupt with *BestCPUIntrClear*.

3 To read data from the CPU port, use *BestCPUportRead* or *BestCPUportWordBlockRead*.

Using *BestCPUportRead*, you can read bytes or words from the CPU. The width of this data is specified by the size parameter.

With *BestCPUportWordBlockRead*, you can read a data block of words.

NOTE For reading words, the address must be word aligned.

4 To write data to the CPU port, use *BestCPUportWrite* or *BestCPUportBlockWrite*.

Using *BestCPUportWrite*, you can write bytes or words to the CPU. The width of this data is specified by the size parameter.

Using *BestCPUportWordBlockWrite*, you can write a data block of words.

NOTE For writing words, the address must be word aligned.

5 To disable the CPU port, set the CPU port mode to “disabled”. This sets the outputs to high impedance.

Use *BestCPUportPropSet*.

6 Furthermore, you can control the RST# line of the CPU port.

Use *BestCPUportRST*.

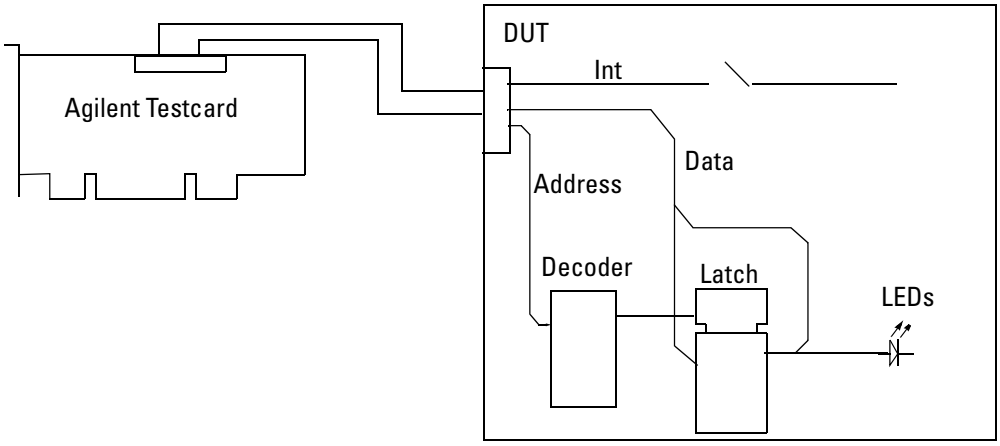
Example

Task The CPU port of the testcard is to drive 8 LEDs on an example device.

To do this, the following is necessary:

- Interrupts have to be generated manually using a switch with a debouncing circuit.
- If the switch is actuated, the program should read an 8-bit value from the device, increment it and write the new value back to the device.

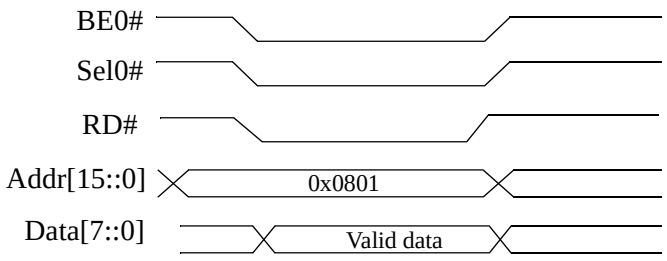
Example Device:



Implementation

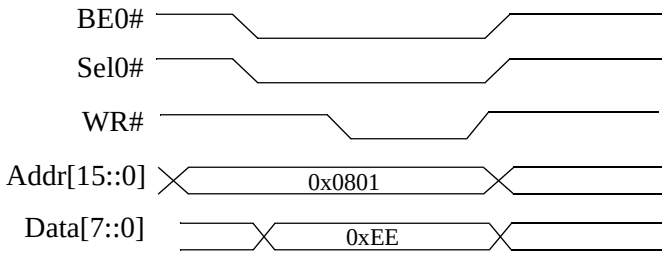
```
/* Enable the CPU port by setting the CPU port mode to "master".
For the remaining port properties, use the default values. */

err=BestCPUportPropSet(handle, B_CPU_MODE, B_CM_MASTER); C(err);

/* Wait for an interrupt by reading the interrupt status of the CPU
port until an interrupt is recognized. */

while(!intr)

{

   err=BestCPUIntrStatusGet(handle, &intr); C(err);

}

/* Clear the interrupt. */
err=BestCPUIntrClear(handle); C(err);

/* Read a byte from the CPU port address 0x0801. */
err=BestCPUportRead(handle, 0, 0x0801, &data, B_SIZE_BYTE); C(err);
```

The figure below shows the respective timing diagram:




```
/* Increment and rewrite data to the same CPU port address. (The
data width must be specified by the size parameter.) */

data++;

err=BestCPUportWrite(handle, 0, 0x0801, data, B_SIZE_BYTE); C(err);
```

The figure below shows the respective timing diagram:



Static I/O Port Programming

The static I/O port can transfer data (for example, status information) between the testcard and the test environment during execution of a test.

Each of the static I/O port pins can be programmed as either:

- input
- totem-pole output
- open-drain output

This involves the following:

- The testcard sets the pin properties (such as specifying input/output pins).
- The testcard reads or writes the data.

Connecting the Static I/O Port to a Decoder

The static I/O port is an internal resource that can be connected to a decoder so that the testcard's target can be used for direct transfers between CPU port and PCI bus. Do this according to the following rules:

- The decoder properties must be set as follows:
  - "Resource" to "Static I/O"
  - "Resource size" to 2
  - "Base address" to 0
- Only words can be transferred.
- The upper byte of the word must contain the output enables for each pin.
- The lower byte must contain the data bits to be transfer.

The static I/O pins provide two output modes:

- **Totem pole**

  The output enable bit must be 0.

  The data bit is driven to the static I/O pin.

- **Open drain**

  The data bit must always be 0. It is driven by the external circuit.

  The output enable bit must be 0 to set the static I/O pin to "active", or 1 to set it to "high impedance".

If a decoder and software use the static I/O port simultaneously, this can result in conflicts. It is not recommended. Note that locking the static I/O port locks the static I/O port property settings but not the transfers.

For more information on decoders, refer to *"Programming the Target Decoder Properties Memory" on page 109*.

Functions Overview

Programming Steps Programming the static I/O port requires the following steps:

1 To specify input/output pins, set the respective pin property.

Use *BestStaticPropSet*.

NOTE Repeat this for each pin to be configured other than input. (By default, all pins of the static I/O port are input.)

2 To read data, a complete byte must be read from static I/O port.

Use *BestStaticRead*.

NOTE Reading a single pin is not supported.

3 To write a bit to a pin of the static I/O port, or to invert it for a period of time, use *BestStaticPinWrite*.

To write a byte to the static I/O port, use *BestStaticPinWrite*.

Example

Task Configure pin 2 of the static I/O port as a totem-pole output and write a value of 1 to this pin.

Implementation

```
/* Configure pin 2 as a totem-pole output. */
err=BestStaticPropSet(handle, 2, B_STAT_PINMODE, B_PMD_TOTEMPOLE);
C(err);

/* Write a value of 1 to pin 2. */
err=BestStaticPinWrite(handle, 2, 1); C(err);

/* Alternatively, write a complete byte to static I/O port or read
a complete byte from static I/O port. */
for(i = 0; i < 4; i++) \
   err=BestStaticPropSet(handle, \
                         i, \
                         B_STAT_PINMODE, \
                         B_PMD_TOTEMPOLE); C(err);

for(i = 4; i < 8; i++) \
   err=BestStaticPropSet(handle, \
                         i, \
                         B_STAT_PINMODE, \
                         B_PMD_INPONLY); C(err);

err=BestStaticWrite(handle,0x0F); C(err);
err=BestStaticRead(handle, &value); C(err);

printf("Value input at static IO port (pins 7:4) %i", value>>4);
```

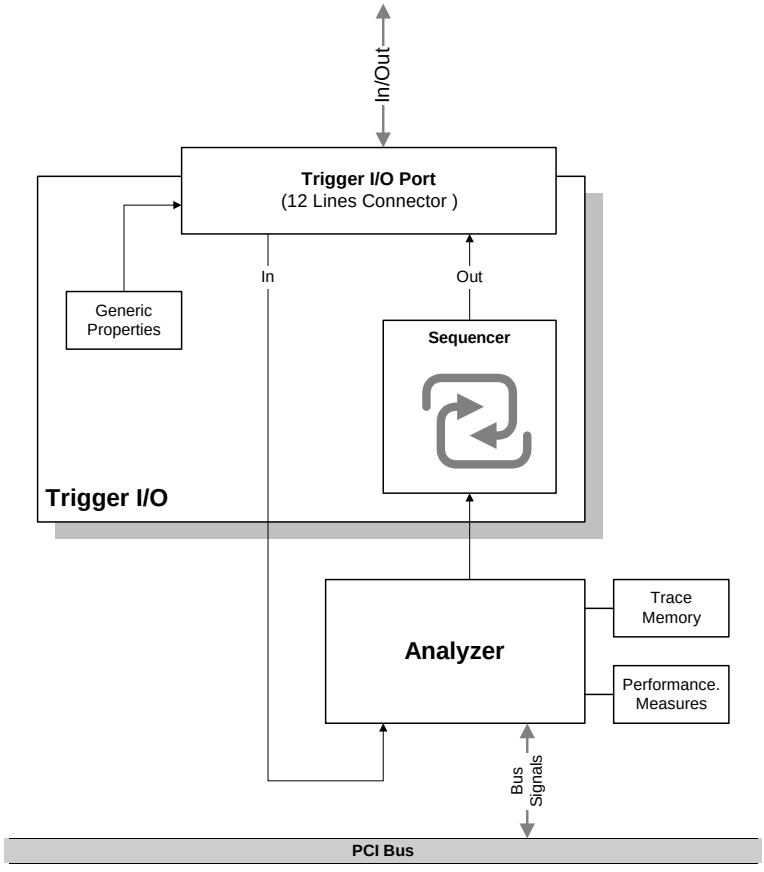
Trigger I/O Sequencer Programming

The Agilent E2925B testcard provides a connector with 12 external trigger lines (trigger port). The trigger lines can be used to synchronize the testcard and other parts of the test environment on the basis of the PCI clock.

Each trigger line can be programmed as either input, open-drain output, or totem-pole output:

- Input trigger signals can be used in all pattern terms.
- Output trigger signals are generated by the trigger sequencer.

The following figure gives an overview of the trigger input and output.



By programming the generic properties, you can specify which lines of the trigger I/O port are input and which are output.

The **output** lines of the trigger port are controlled by a programmable sequencer. For more information on sequencers, refer to *"Sequencer Programming" on page 55*.

The trigger I/O **input** lines are directed to the analyzer and can there, for example, be evaluated by triggering the trace memory or, being counted, or stored in the trace memory.

Functions Overview

The Agilent E2925B testcard's programming interface provides functions for programming generic properties and the sequencer. The available functions and their usage is shown by describing the programming steps.

Programming Steps

Programming the trigger I/O sequencer requires the following steps:

1. Set the preload value for feedback counter C.

   Use *BestTrigIOSeqGenPropSet*.

   With this function, you can also determine the output mode of trigger line 0 … 11.

2. Set all properties in the trigger I/O sequencer description table to default values.

   Use *BestTrigIOSeqPropDefaultSet*.

3. 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 must turn true at a time. Otherwise, the software will not accept the table because the table does not uniquely define the sequencer's behavior.

Use *BestTrigIOSeqTranPropSet*.

4 Set conditions in the trigger I/O sequencer description table. Conditions can be:

   – transition condition for the trigger I/O sequencer to move from one state to the next

   – output conditions on trigger I/O pins

   – conditions to decrement or preload the feedback 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 referring to pattern terms (`pt0`, `pt1`, ...) and the terminal count (`tc`) of the feedback counter C.

   Use *BestTrigIOSeqTranCondPropSet*.

5 Write the trigger I/O sequencer description table to the trigger I/O sequencer memory.

   Use *BestTrigIOSeqProg*.

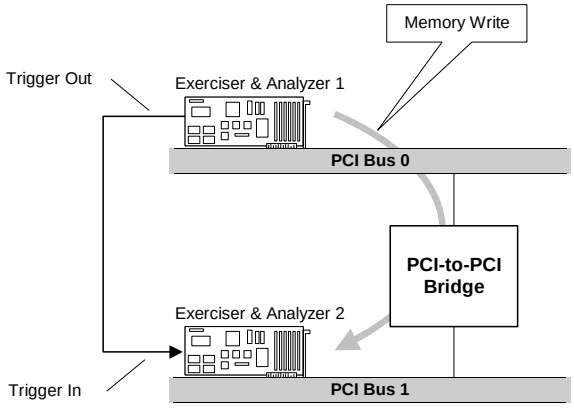
6 Start the trigger I/O sequencer.

   Use *BestTrigIORun*.

7 To stop the trigger I/O sequencer, use *BestTrigIOStop*.

Example

Task To measure the latency of a PCI-to-PCI bridge, the trigger is to be programmed. The following figure shows the example test system:



In the example, two testcards are needed. The testcards are placed on two PCI buses that are connected via the PCI-to-PCI-bridge under test. The testcards are set up for data transfer from one to the other, for example, with the Memory Write command.

The *trigger output* of the testcard 1 (master) is connected to a *trigger I/O* input line of testcard 2 (target). Testcard 1 gives a pulse to testcard 2 via this connection when it starts the transaction. This pulse triggers the testcard 2's trace memory.

To accomplish this, you need to program testcard 1 to transfer data and send the pulse to testcard 2. To set up testcard 1, you need to program the trigger I/O sequencer for trigger out as follows:

- **Pattern Terms**

  To recognize the transfer, pattern term 1 must be sensitive to master transactions:

  ```
  pt1 == "m_xact"
  ```

- **States**

  A pulse is issued on trigger I/O line 0 when a transaction begins. Afterwards the line returns to 0. This can be done with the following sequencer description table:

Transient No.	State	Next State	Transition Condition	Trigger Line #0 (Output)
0	0	0	!pt1	0
1	0	1	pt1	1
2	1	1	pt1	0
3	1	0	!pt1	0

Implementation

```
/* Enable trigger out line 0 as totem-pole output.*/
err=BestTrigIOGenPropSet(handle, \
                         B_TRIGIOSEQGEN_OUT_0, \
                         B_TRIGIO_TOTEMPOLE); C(err);


/* Set pattern term 1 to detect master transactions. */
err=BestPattSet(handle, B_PATT_TERM_1, "m_act"); C(err);


/* Initialize the trigger I/O sequencer description table. */
err=BestTrigIOSeqPropDefaultSet(handle); C(err);
```

```
/* Initialize and set up transient 0. */
err=BestTrigIOSeqTranPropDefaultSet(handle, 0); C(err);
err=BestTrigIOSeqTranPropSet(handle, 0, B_TRIGIOSEQ_STATE, 0);
C(err);
err=BestTrigIOSeqTranPropSet(handle, 0, B_TRIGIOSEQ_NEXTSTATE, 0);
C(err);
err=BestTrigIOSeqTranCondPropSet(handle, 0, B_TRIGIOSEQ_XCOND,
"!pt1"); C(err);
err=BestTrigIOSeqTranCondPropSet(handle, 0, B_TRIGIOSEQ_OUT_1,
"0"); C(err);


/* Initialize and set up transient 1. */
err=BestTrigIOSeqTranPropDefaultSet(handle, 1); C(err);
err=BestTrigIOSeqTranPropSet(handle, 1, B_TRIGIOSEQ_STATE, 0);
C(err);
err=BestTrigIOSeqTranPropSet(handle, 1, B_TRIGIOSEQ_NEXTSTATE, 1);
C(err);
err=BestTrigIOSeqTranCondPropSet(handle, 1, B_TRIGIOSEQ_XCOND,
"pt1"); C(err);
err=BestTrigIOSeqTranCondPropSet(handle, 1, B_TRIGIOSEQ_OUT_1,
"1"); C(err);


/* Initialize and set up transient 2. */
err=BestTrigIOSeqTranPropDefaultSet(handle, 2); C(err);
err=BestTrigIOSeqTranPropSet(handle, 2, B_TRIGIOSEQ_STATE, 1);
C(err);
err=BestTrigIOSeqTranPropSet(handle,  2, B_TRIGIOSEQ_NEXTSTATE, 1);
C(err);
err=BestTrigIOSeqTranCondPropSet(handle, 2, B_TRIGIOSEQ_XCOND,
"pt1"); C(err);
err=BestTrigIOSeqTranCondPropSet(handle, 2, B_TRIGIOSEQ_OUT_1,
"0"); C(err);


/* Initialize and set up transient 3. */
err=BestTrigIOSeqTranPropDefaultSet(handle, 3); C(err);
err=BestTrigIOSeqTranPropSet(handle, 3, B_TRIGIOSEQ_STATE, 1);
C(err);
err=BestTrigIOSeqTranPropSet(handle, 3, B_TRIGIOSEQ_NEXTSTATE, 0);
C(err);
err=BestTrigIOSeqTranCondPropSet(handle, 3, B_TRIGIOSEQ_XCOND,
"!pt1"); C(err);
err=BestTrigIOSeqTranCondPropSet(handle, 3, B_TRIGIOSEQ_OUT_1,
"0"); C(err);


/* Write the sequencer description table to the sequencer memory.
The transition conditions are checked for consistency. */
err=BestTrigIOSeqProg(handle); C(err);
```

```
/* Start the trigger I/O sequencer. */
err=BestTrigIORun(handle); C(err);
```

NOTE To complete the test, you additionally need to set up and then run the following:

- Testcard 1 has to be set up as master to transfer data.
  See *"Programming the Exerciser as a Master Device" on page 80*
- Testcard 2 has to be set up as target to receive the data.
  "Programming the Exerciser as a Target Device" on page 105.
- Testcard 2's analyzer has to trigger at the pulse.
  "Programming the Analyzer" on page 45.

LED Controlling and Display Functions Overview

Programming Steps

To control the LED display, the following steps are required:

1 Set the mode of the LED display.

Before writing values to the display, select “user mode”.

Use *BestDisplayPropSet*.

2 Write a value to the LED display.

Use *BestDisplayWrite*.

Example

Task

Write a byte to the hex display.

Implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <mini_api.h>
#include <regconst.h>


#define CHECK  { if(status != B_E_OK) \
 { printf("%s\n", BestErrorStringGet(status)); return -1; } }


int main (int argc, char *argv[] )
{
b_errtype status;
b_handletype handle;
b_int32 devid;
int i,j;


/* Get device number. The subsystem id (0 in this example) can be
used to distinguish between multiple testcards*/
printf("getting devid\n");
getchar();


/*Initialize port internal structs and variables.*/
printf("Opening Best\n");
status = BestDevIdentifierGet(0x103c, 0x2925, 0, &devid); CHECK
status = BestOpen(&handle, B_PORT_PCI_CONF, devid); CHECK
```

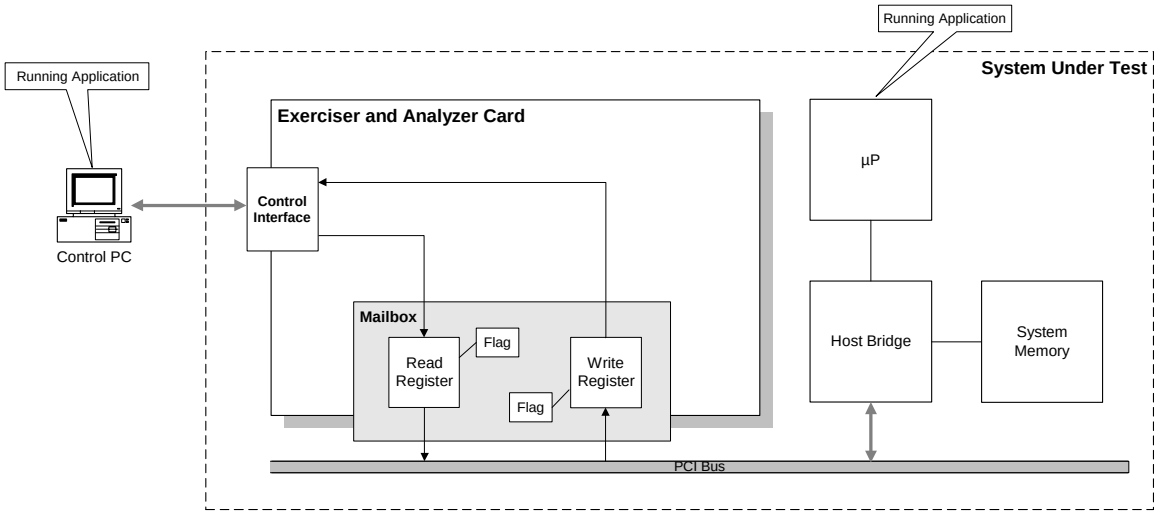
```
/*Establish connection to testcard.*/
printf("Connecting to Best\n");
status = BestConnect ( handle ); CHECK;


/* Application program goes in here, for example:*/

/* Put hex display into user mode */

status = BestDisplayPropSet(handle, B_DISP_USER); CHECK
for(i = 0; i < 2000; i++)

{
    for(j = 0; j < 100000; j++);

    /* Write byte to hex display */
    status = BestDisplayWrite(handle, i%256); CHECK

}

/* Put hex display into protocol observer mode */
status = BestDisplayPropSet(handle, B_DISP_CARD); CHECK

/* Disconnect from the current port.*/
status = BestDisconnect (handle); CHECK;


/* Close the session and deallocate memory*/
status = BestClose(handle); CHECK;

return 0;

}
```

Mailbox Programming

The mailbox of the Agilent E2925B 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 parallel port (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. 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 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 Access to the Mailbox

The mailbox registers are located in the private section of the Agilent E2925B 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 after access to the other bytes.

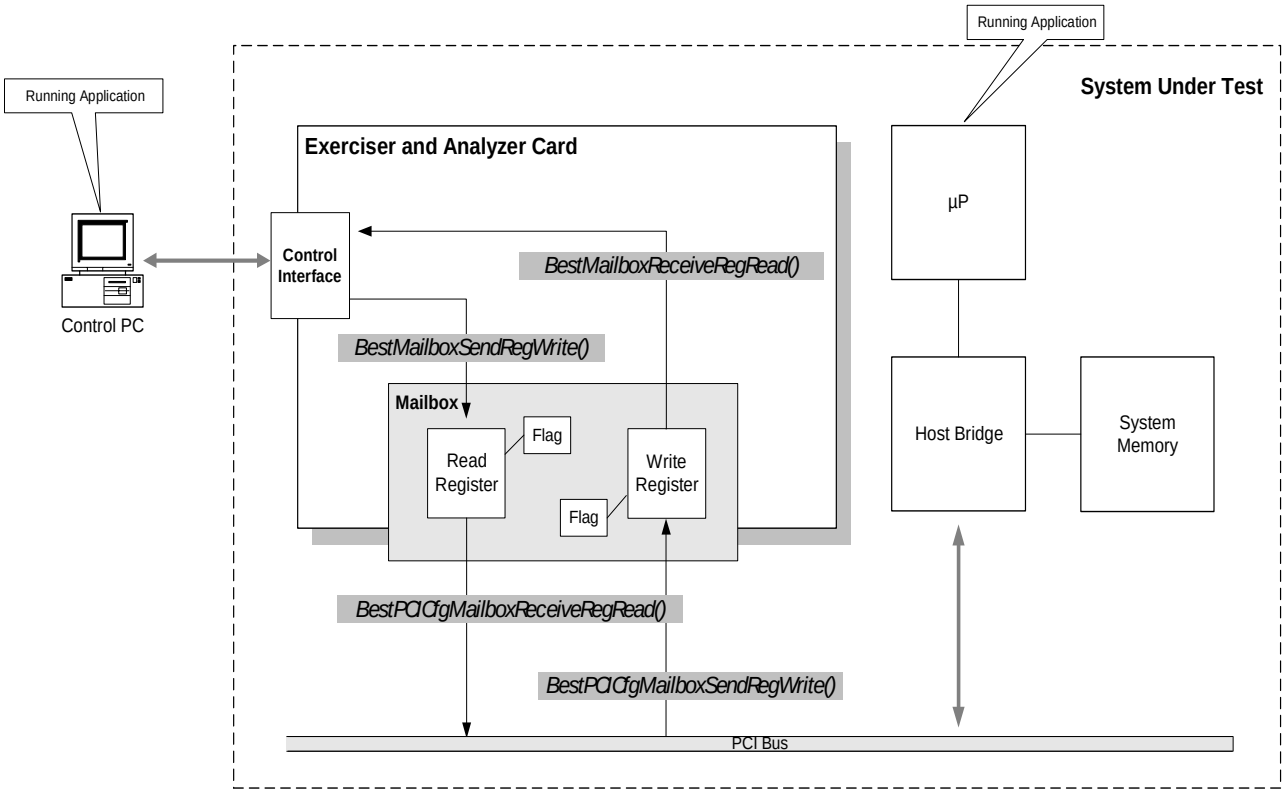
Mailbox Status Register

The following table shows the mailbox status register in the configuration space:

Offset Config	Bits	Type	Oper.	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 to inform the CPU that the mailbox register has been read and clears the flags.

Functions Overview

The following figure shows the available mailbox functions and the application:



Programming Steps for Access via PCI Bus

To access the mailbox via PCI bus, the following steps are required:

1. Identify the testcard.

   Use *BestDevIdentifierGet*.

   Because multiple PCI testcards can be plugged into the system under test, the Agilent E2925B testcard needs to be identified for mailbox access.

2. To write data to the mailbox via the PCI bus, use *BestPCICfgMailboxSendRegWrite*.

   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. Use *BestPCICfgMailboxReceiveRegRead*.

Programming Steps for Access via Control PC

To access the mailbox via the control PC:

- Send and receive data using the control interface.

  Use *BestMailboxSendRegWrite* and *BestMailboxReceiveRegRead* respectively.

Example

The following code fragments give examples of the two ways of accessing the mailbox:

Task Write data to the mailbox via the PCI bus.

Implementation

```
/* Identify the testcard and write data to the mailbox until the
flag indicates that data has been written successfully. */
err=BestDevIdentifierGet(0x103C, 0x2926, 0, &devid); C(err);

do {
   err = BestPCICfgMailboxSendRegWrite(devid, data, &status);
   C(err);
} 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 {
   err = BestMailboxReceiveRegRead(handle, &data, &status); C(err);
} while(status == 0);
```

Power Management Event Programming

The Agilent E2925B testcard can control PCI Power Management Events (PME). The events are used by a PCI device to “wake up” the PC if it is in power save mode.

Power Management Event Functions

The C-API provides functions for reading and writing on the PME line. This allows the testcard to emulate both the PC receiving a power management event, and the device issuing it.

- To read the PME line, use *BestPMERead*.
- To write on the PME line, use *BestPMEWrite*.

Example

The following line issues a power management event to the PCI system, so that it wakes up from power save mode, or winds up power save timers:

```
err=BestPMEWrite(handle, 1);C(err);
```

Using the PPR

The following sections describe the PCI Permutator and Randomizer software and show how to use it.

- *"Generating Permutations" on page 178* gives basic information about permutations supported with the PPR software.
- *"Example Test Design" on page 183* shows a typical scenario to be tested. This example is subject of all further sections.
- *"How to Write a Test Program" on page 182* introduces the steps required for setting up a test program.
- *"PPR Administration" on page 186* gives detailed information on the first steps for setting up a test program.
- *"Programming Master Block Permutations" on page 189* gives detailed information on generating and programming master block permutations.
- *"Programming Master Attribute Permutations" on page 199* gives detailed information on generating and programming master attribute permutations.
- *"Programming Target Attribute Permutations" on page 205* gives detailed information on generating and programming target attribute permutations.
- *"Generating PPR Reports" on page 209* gives information on the contents of a PPR report and shows how to program it.
- *"Running the PPR Test" on page 211* shows the required programming steps for running a test and checking for protocol errors.
- *"Analyzing the Report" on page 213* describes all information generated in a PPR Report.
- *"Further Options and Possibilities" on page 229* shows how to optimize the testing time and to avoid unexpected program behavior, and informs about byte enable variation, a more exhaustive test, uncovered permutation and reproducing bus errors.
- *"Report Listing" on page 232* shows the complete C program and the complete report for the example specified in *"Example Test Design" on page 183*.

Generating Permutations

If you are interested in performing tests using PCI Permutator and Randomizer software, you need a general understanding of the algorithms, and you should know some basic terms.

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**:

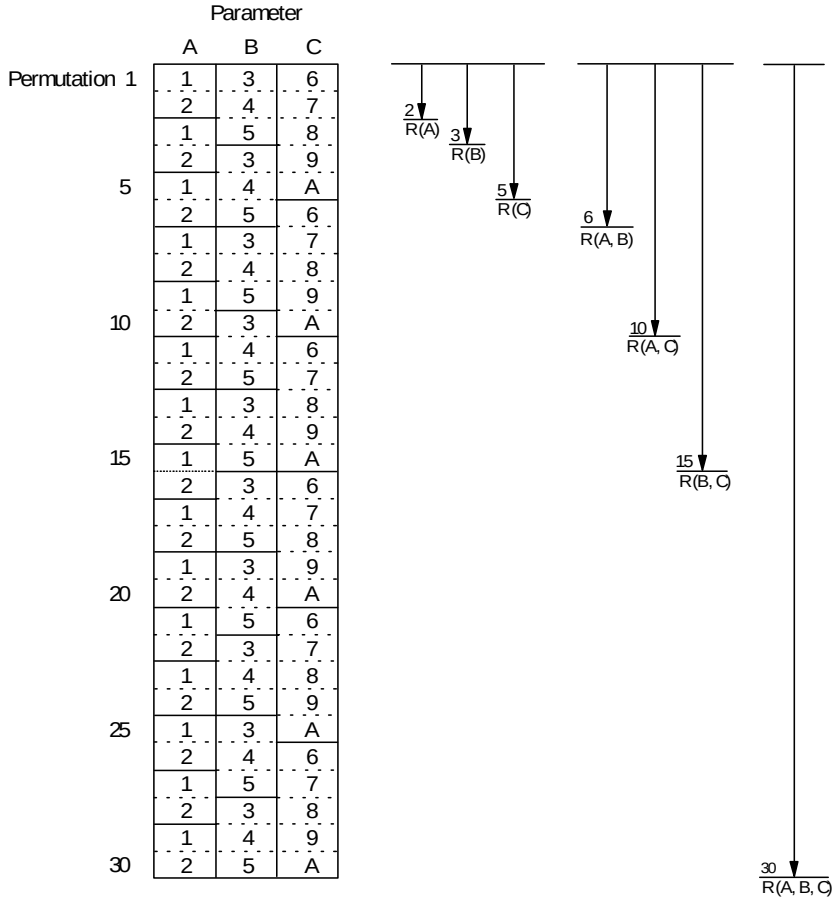
- Parameter A can take the following 2 values: 1 and 2.
- Parameter B can take the following 3 values: 3, 4 and 5.
- Parameter C can take the following 5 values: 6, 7, 8, 9 and A.

Different strategies can be pursued to combine each value of a parameter with all values of the other parameters at least once.

The PCI 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.



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.

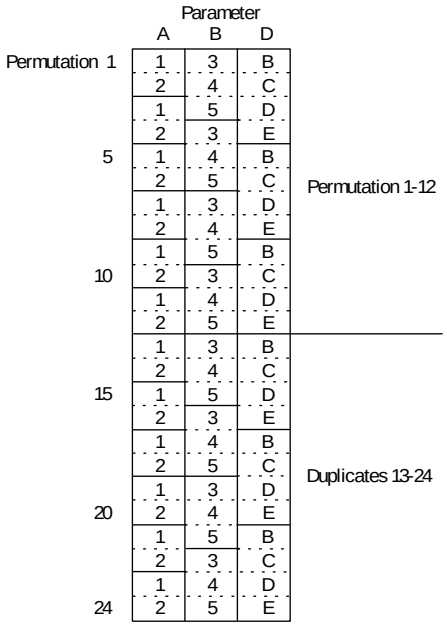
This is represented by the *"Permutation Table" on page 179*. The 31st tuple would again be the same as tuple 1, the 32nd tuple as tuple 2, and so on. This means, that all possible combinations of the values of A, B and C are covered after 30 permutations (*coverage*=30).

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.



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 Protocol Permutator and Randomizer software.

Programming Steps

Writing a C program 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 *"Error Checking" on page 20*.

2. Initialize the software.

   This is done by setting generic properties. See *"PPR Administration" on page 186*.

3. Program block variation permutations.

   See *"Programming Master Block Permutations" on page 189*.

4. Set up master attribute permutations.

   See *"Programming Master Attribute Permutations" on page 199*.

5. Set up target attribute permutations.

   See *"Programming Target Attribute Permutations" on page 205*.

6. Set up the report properties.

   For setting up the properties and printing the report, see *"Generating PPR Reports" on page 209*

7. Run the test.

   See *"Running the PPR Test" on page 211*.

8. Set up the program footer.

   Deinitialize the PCI Permutator and Randomizer software and terminate the BEST software. See *"PPR Administration" on page 186*.

A complete reference of the available functions can be found in *Agilent E2925B Opt. 320 C-API/PPR Reference*.

Example Test Design

To illustrate the basic concepts of the Permutator and Randomizer, this section shows an example test. 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 can be found in the exerciser's internal memory, beginning with line 0. The block is to be transferred to a memory block in the system memory, beginning with the starting address `B8000\h`. The system is assumed to provide a 32-bit PCI bus.

During the transfer, the following protocol variations should occur:

Variations	Variation Parameter	Allowed Values
Block Variations	address alignments	(%16=0) (%16=4) (%16=8) (%16=12) (%32=0)
	blocksizes (in bytes)	4 8 16
	bus commands	7 = Memory Write 15 = MWI
Master Attributes	last (burstlength) group: ML	4 8 32
	waits group: MD0	0 1 3 8
	steps group: MA1	0 7
	tryback group: MA1	true false

Permutations to be Covered

Different testing areas must be covered during the example test:

- Block Variation Permutations (testing area BLOCK)

  The following permutations of block variation parameters (address alignments, blocksizes, bus commands) are required:

  – each address alignment must occur

  – each blocksize must occur

  – each blocksize must start at each address alignment at least once

  – transfers must be executed with and without `MWI` (with `MWI`, if it is possible)

- Master Attribute Permutations (testing area MATTR)

  The following permutations of master attributes (burstlengths, waits, steps, tryback) are required:

  – each burstlength must occur

  – each count of wait cycles must occur

  – each count of steps must occur

  – transfers must occur with and without tryback

  – each count of wait cycle must be combined with each burstlength

  – steps must be combined with tryback to ensure that back-to-back is tried at least once

- Testing Area ALL

  For this testing area, each block variation permutation must meet each master attribute permutation at least once.

The following table summarizes the permutations required for this example.

Testing Area	Requirements	Tuple
BLOCK	All address alignments occur.	(alignment)
	All blocksizes occur.	(blocksize)
	Transfers with and without MWI .	(commands)
	Blocks of all sizes start at all alignments.	(blocksize, alignment)
MATTR	All burstlengths occur.	(burst)
	All numbers of wait cycles occur.	(wait)
	Steps 0 and 7 must occur.	(steps)
	Tryback occurs at least once or not at all.	(tryback)
	All counts of wait cycles meet all positions of all bursts of 4, 8, and 32 dwords.	(burst, wait)
	Steps = 0 must meet tryback = 1 at least once.	(steps, tryback)
ALL	All desired block permutations meet all desired master attribute permutations.	(BLOCK, MATTR)

Resource Constraints

Resource constraints are determined by the resources of the PCI Exerciser and Analyzer testcard available at the moment the test is run. For this example the following is assumed:

- A maximum of 60 blocks may be allocated.

  This means that the master block page size can hold a maximum number of 60 blocks (Master Block Page Size MBPS = 60).

- A maximum of 49 lines in the master attribute page may be allocated.

  This means that the Master Attribute Page Size (MAPS) must be less than 49 attribute lines.

- The total testing time must be less than 100 ms, neglecting an initial programming overhead.

  Once specified, the programming overhead and other system parameters are reused and do not need to be reinitialized.

- The compound blocksize (CBS) is 64 dwords = 256 bytes. (This value should always be a power of two.)

- The cacheline size is 16 bytes (4 dwords).

  The cacheline size of 16 bytes has been chosen in order to keep the example short. Modern systems of today have a cacheline size of 32 bytes.

A more exhaustive test would use nearly all hardware resources of the exerciser. This test case is more closely considered in *"Further Options and Possibilities" on page 229*.

NOTE How to program the desired permutations and how to get the test results can be found in the following sections.

The complete implementation of this example can be found under *"Example: Using the PPR" on page 23*.

PPR Administration

Before the PCI PPR software can be used, the testcard and its connections have to be initialized. See *"Programming the Framework" on page 27*.

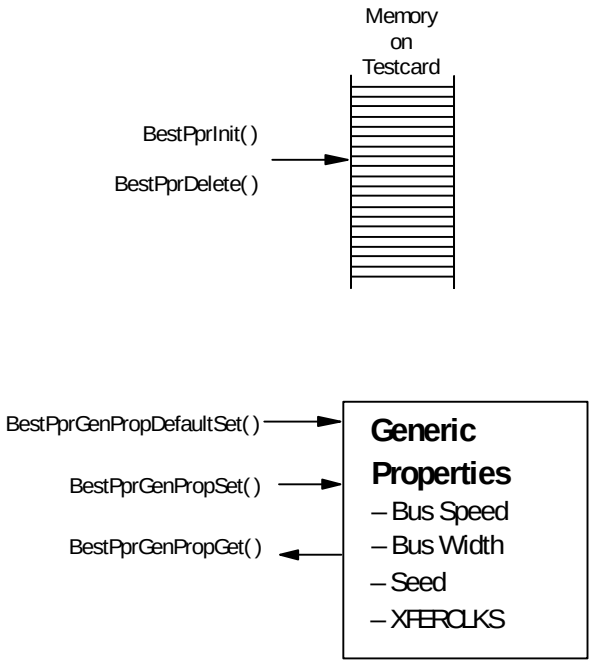
After the Exerciser and Analyzer software has been initialized, the PPR software must be initialized and generic properties, such as bus speed or bus width, can be set.

After using the PPR software, the allocated memory must be freed before the Exerciser and Analyzer software is terminated.

See *"Functions Overview" on page 187*.

Functions Overview

The following figure shows the generic setup functions used to initialize and deinitialize the PCI Protocol Permutator and Randomizer software and for getting and setting general properties.



Programming Steps

Setting up the test program requires the following steps:

1. Initialize the Exerciser and Analyzer testcard.

   See *"Connection and Initialization" on page 28*.

2. Initialize the PPR software by setting all properties of this software to default values.

   Use *BestPprInit*.

3. Set general properties, such as PCI bus speed and bus width, the expected number of clocks per data transfer and a random seed.

   Use *BestPprGenPropDefaultSet* and *BestPprGenPropSet*.

   Use *BestPprGenPropGet* to read the settings.

4. At the end of the test, free all memory allocated by the software.

   Use *BestPprDelete*.

Example

Task Perform the general setup for your test program as follows:

- The master attributes should cycle through their values sequentially, independent of whether or not a new master block page starts. To perform this, set the corresponding general master property.
- The system is assumed to provide a 32-bit PCI bus.

Implementation

```
/* Request the session handle needed in all following functions and
open the connection to the control PC via Fasthost interface. */
   status = BestOpen( &handle, B_PORT_PARALLEL,
   B_PORT_LPT1); CHECK;
/* Initialize the software by specifying a general property so that
the master attributes cycle through their values sequentially,
independent of whether or not a new master block page starts. */
   status = BestMasterGenPropSet( handle,
                                  B_MGEN_ATTRMODE,
                                  B_ATTRMODE_SEQUENTIAL); CHECK;

/* Initialize the PCI Protocol and Randomizer software. */
   status = BestPprInit( handle ); CHECK;

/* Set the generic PPR property buswidth to 32 bit. */
   status = BestPprGenPropSet( handle, BPPR_GEN_BUSWIDTH, 32 );
   CHECK;

/* Insert the application program here. */
/* … */

/* Deinitialize the PCI Permutator and Randomizer software. */
    status=BestPprDelete( handle ); CHECK;

/* Terminate the BEST software. */
    status=BestClose( handle ); CHECK;
}
```

Programming Master Block Permutations

This section describes how a block transfer is prepared and explains the properties important for the block transfer and the permutations.

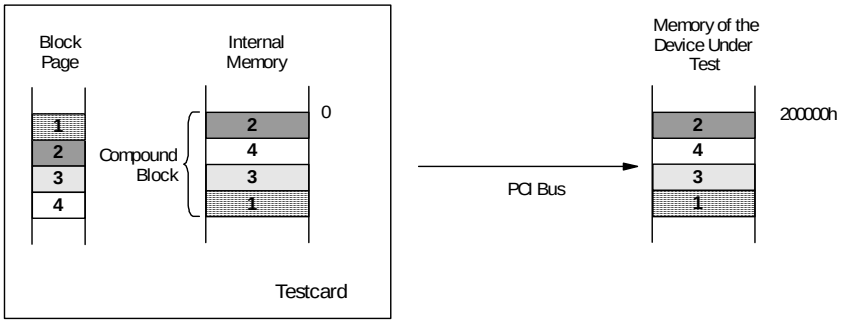
NOTE For more information on blocks, please refer to *"Master Block Transfer Memory Programming" on page 85*.

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 master and may require multiple bursts to complete, due to the master's intention, target termination, or an intervention of the arbiter.

Compound Block The PCI Permutator and Randomizer software combines a multiple number of blocks into a **compound block**, which contains a series of block transfers. The blocks reside within a contiguous range in memory, for example corresponding to the memory range of the system under test (or a part of its memory range).

Block Page The individual blocks of the compound block can be executed in any order. The information on the order in which the block transfers are performed is contained in a **block page** (a page of the block transfer memory) on the testcard.

The following figure shows an example of the memory transfer concept. On the testcard, there is a block page and a compound block of 4 blocks, prepared for transfer from the testcard's internal memory to the memory of the system under test.



The PCI Protocol Permutator and Randomizer software internally permutates the different variation parameters of these blocks against each other to compute the required block page size, the last possible permutation using the specified block page size, and the estimated testing time. On demand, the block page can then be generated and downloaded to the testcard. The testcard can generate the traffic by executing that block page.

Block permutation properties and variation parameters are described in the following subsections.

Block Permutation Properties

Block permutation properties define the intention of the compound block. The following properties can be set:

Transfer Direction The transfer direction is seen from the master's side:

- *write* from internal memory to system memory
- *read* from system memory into internal memory

Compound Block Size The compound blocksize (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.

NOTE It is recommended that the compound blocksize is set to a power of 2.

Dual Address Cycles Forces the Exerciser and Analyzer to use dual address cycles when used in a 64-bit system.

Bus Address The bus address is the starting address in the PCI memory range of the system under test to which the compound block will be transferred, or from which it will be read.

WARNING Allocate the required memory in your test program. Writing directly into system memory passing by the operating system may cause a serious system crash.

Internal Address The internal address is the starting address in the memory range of the Agilent E2925B testcard to which the compound block will be transferred, or from which it will be read.

NOTE The PCI Protocol and Randomizer software does not fill up memory with data. This can be done by appropriate standard C-API functions. See *"Data Memory and Compare Unit Programming" on page 142*.

Attribute Page This property specifies a master attribute page, which is then used for the compound block transfer.

Compare Flag and Compare Offset If the compare flag is set, a compound block being read is compared with a data block found at the location specified by the compare offset. The result is stored in the testcard's status register.

Refer to "*b_blkproptype*" described in the *Agilent E2925B Opt. 320 C-API/PPR Reference* for detailed information on use and ranges of the compare flag and offset.

Master Block Page Size (MBPS) This property specifies the maximum number of blocks a compound block can contain.

First Permutation Number 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 attribute page.

Fill Gaps 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.

Block Variation Parameters

Block variation parameters specify how the compound block is to be intensified by permutated variations of parameters, such as blocksize 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.

For the selection of bus commands, special algorithms are available. These algorithms either select the best suitable command or the next command in the list, taking into account recommendations of the PCI specification and always considering the other variation parameters.

For more information about these algorithms, refer to *"bppr_algorithmtype"* in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

The following block variation parameters are available:

Start Address Alignment A list of arbitrary start address "alignments" can be specified. An alignment consists of a value for address granularity and a value for the offset within this granularity. This allows block transfers to start at certain offsets relative to certain address granularities, for example, start one dword after a 32-byte boundary (`%32=4`).

The granularity could, for example, correspond to the cacheline size of the system under test.

Block Size A list of blocksizes can be specified in bytes.

Byte Enables A list of byte enables can be specified to occur in the data phase of the block transfer.

If some byte enables are not active in one block and the "Fill Gaps" property is set, another block will be transferred with these byte enables set active.

Bus Commands A list of PCI bus commands can be specified. All PCI bus commands may be specified, but only those commands that are suitable for the specified transfer direction will be used for variations.

The use of the `MWI` (Memory Write and Invalidate) command is restricted by the PCI specification. Optionally, the use of the extended memory commands MRL (Memory Read Line) and MRM (Memory Read Multiple) can also be restricted to PCI recommendations.

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 master block permutation depends on the number of variation parameters examined, the PCI 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 containing an extended command.

	Algorithm			
Direction	**RAND**	**PERM**	**RECOMM**	**BEST**
READ	No coverage can be guaranteed	Coverage = Repetition length of commands in the list, raised up to the next prime	Coverage = Repetition length of the **recommended** commands in the list, raised up to the next prime × Repetition length of the influencing parameters, raised up to the next prime	Coverage = Repetition length of the influencing parameters, 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		

Available Algorithms

The software provides the following algorithms:

RAND The randomizing algorithm picks commands from the list at random without eliminating duplicate tuples. Therefore coverage can never be guaranteed. The `MWI` command is replaced by the “memory write” command (MW), if it had to be used in an invalid parameter combination.

PERM The permutating algorithm picks one command after the other from the list and combines them with the other parameters, regardless of whether the command is suitable or not. The `MWI` command is replaced by the “memory write” command (MW), if it had to be used in an invalid parameter combination.

RECOMM This algorithm combines a parameter combination only with commands recommended by the PCI specification.

BEST This algorithm combines only the best suitable command with the parameter combination. This effect is the same as if the list only contained one command.

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 For basic information see *"Generating Permutations" on page 178*.

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 calculated by multiplying the number of values of each property. For example, to combine 5 address alignments with 3 blocksizes, $5 \times 3 = 15$ data transfers (blocks) are required.

If the bus command variations contain extended memory commands, their values must permutate against each block variation parameter. The software considers this by calculating the repetition length R from the variation list lengths N of both these commands *and* the referring block variation parameters. The repetition length can be reduced by selecting the "best" algorithm, which picks only the best command (according to PCI specification) from the value list and ignores all the remaining ones.

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).

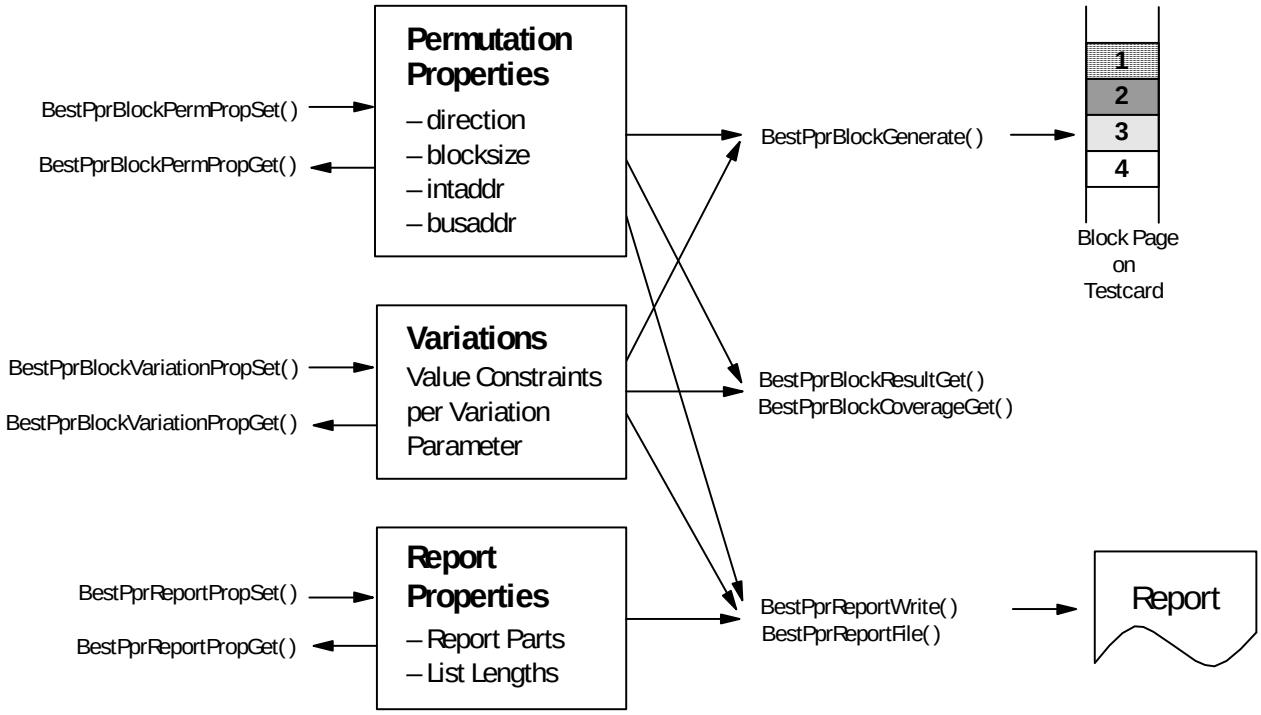
Testing Time

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
Master Block Permutation Testing Time	CBS × Time-per-data-transfer

Functions Overview

The following figure shows the master block and the report functions used to prepare and to perform a master block permutation.



Programming Steps Programming master block permutations requires the following steps:

1. Prepare a permutation by setting the block permutation properties (for example, transfer direction, bus address and internal address).

   Use *BestPprBlockPermPropSet*.

2. Define lists of values for the variations and select the algorithm used to pick the values from this list.

   Use *BestPprBlockVariationPropSet*.

   These lists specify values for the block variation parameters to be permutated according to the testing requirements. Block variation parameters are alignment, blocksize, 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 or coverage.

   Use *BestPprBlockResultGet* and *BestPprBlockCoverageGet*.

4 Perform the permutation.

There are two ways to do this:

– A master block page can be generated and downloaded to the testcard where it can be run using the standard C-API functions.

Use *BestPprBlockGener*ate and *BestMasterBlockPageRun*.

– The permutation results can be requested from the PCI Protocol Permutator and Randomizer software. Adjusted properties and permutation results can then be written to a report file.

Use *BestPprReportWrite* or *BestPprReportFile*.

NOTE The contents of the report file can be controlled with *BetPprReportPropSet* and *BetPprReportPropGet*.

Example

Task Program master block permutations as follows:

- Prepare the block transfer by setting the following block permutation properties:
  - The compound block can be found in the exerciser's internal memory, beginning with line 0.
  - The block is to be transferred to a memory block in the system memory, beginning with the starting address `B8000\h`.
  - A maximum of 60 blocks may be allocated.

    This means that the master block page size can hold a maximum number of 60 blocks (Master Block Page Size MBPS = 60).
  - Master attribute page 2 is to be used.
  - The total testing time must be less than 100 ms, neglecting an initial programming overhead.

    Once specified, the programming overhead and other system parameters are reused and do not need to be reinitialized.
  - The compound blocksize (CBS) is 64 dwords = 256 bytes. (This value should always be a power of two.)
  - The cacheline size is 16 bytes (4 dwords).

    The cacheline size of 16 bytes has been chosen in order to keep the example short. Modern systems of today have a cacheline size of 32 bytes.

- Define the following value lists for block variations that should occur during the transfer, and select the algorithm that picks all values one after the other as listed in the value list:

Variations	Variation Parameter	Allowed Values
Block Variations	address alignments	(%16=0) (%16=4) (%16=8) (%16=12) (%32=0)
	blocksizes (in bytes)	4 8 16
	bus commands	7 = Memory Write 15 = MWI

- Ensure that the following permutations are covered:

Testing Area	Requirements	Tuple
BLOCK	All address alignments occur.	(alignment)
	All blocksizes occur.	(blocksize)
	Transfers with and without MWI .	(commands)
	Blocks of all sizes start at all alignments.	(blocksize, alignment)

Implementation

```
/* Set the transfer direction to "write". */
   status = BestPprBlockPermPropSet( handle,
                                     BPPR_BLK_DIR,
                                     BPPR_DIR_WRITE ); CHECK;

/* Set the bus address to B8000\h. */
    status = BestPprBlockPermPropSet( handle,
                     BPPR_BLK_BUSADDR,
                     0x0b8000 ); CHECK;

/* Set the internal memory first line to 0. */
     status = BestPprBlockPermPropSet( handle, BPPR_BLK_INTADDR, 0
); CHECK;

/* Set the compound blocksize (the number of dwords in the block)
   to 64. */
   status = BestPprBlockPermPropSet( handle, BPPR_BLK_NOFDWORDS, 64 );
   CHECK;

/* Select the master attribute page 2. */
    status = BestPprBlockPermPropSet( handle, BPPR_BLK_ATTRPAGE, 2 );
    CHECK;
```

```
/* Select block page 1, which contains a maximum of 60 blocks. */
    status = BestPprBlockPermPropSet( handle, BPPR_BLK_PAGENUM, 1 );
    CHECK;

    status = BestPprBlockPermPropSet( handle, BPPR_BLK_PAGESIZEMAX, 60);
   CHECK;

/* Set the system cacheline size to 4 dwords. */

 status = BestPprBlockPermPropSet( handle, BPPR_BLK_CACHELINE, 4);
CHECK;

/* Set block variation properties according to the test design
(testing area BLOCK). Each of them should use the algorithm "perm".
(For permutation, this algorithm picks the values from the value
lists in the order in which they appear.) */

/* Specify the alignment values. */

    status = BestPprBlockVariationSet( handle,
                    BPPR_BLK_ALIGN,
                    "(%16=0), (%16=4), (%16=8), (%16=12), (%32=0)",
                    BPPR_ALG_PERM); CHECK;


/* Specify the blocksize values. */

 status = BestPprBlockVariationSet( handle,
                   BPPR_BLK_SIZE,
                   "4,8,16",
                   BPPR_ALG_PERM ); CHECK;


/* Specify the block commands Memory Write (7) and MWI (15). */

    status = BestPprBlockVariationSet( handle,
                     BPPR_BLK_CMDS,
                     "mem_write, mem_writeinvalidate",
                     BPPR_ALG_PERM ); CHECK;


/* After setting up these parameters, generate the block and
download it to the testcard. Note, if you first want to check
whether your test requirements are really suitable, you can request
the test results or coverage. In this case, first skip the
following line, and add it after you have checked and corrected
your test scenario. */

    status = BestPprBlockGenerate( handle ); CHECK;
```

Programming Master Attribute Permutations

The master attributes can be constrained and permutated in the same way as the block variation parameters described above. The result can be downloaded to a master attribute page on the testcard. The size of the master attribute page (MAPS) is determined by the master attribute group with the most memory lines and is selectable within the range limited by the exerciser.

Master Attributes

To achieve more sophisticated randomization opportunities, the master attributes are divided into groups, which are varied against each other. The following tables show which attributes are assigned to which group:

Group	Address Phase Attributes
MA0	Delay (Exerciser Idle)
MA1	Try "Fast Back-to-Back", Steps
MA2	64 Bit Request, Release Request
MA3	Resume Delay
MA4	Wrong parity signalling, parity and system errors

Group	Data Phase Attributes
MD0	Waits
MD1	Parity and system errors
MD2	Wrong parity signalling

Group	Control Attributes
ML	Last

Variation Parameters

A list of values and the algorithm for picking the values from the list can be specified for every master attribute.

The burstlength determined by the attribute `Last` of group `ML` is of special interest, because each burst starts with an address phase followed by as many data phases as determined by the burstlength. On the attribute page, the attribute `LAST` is boolean and is set to 1 in the last data phase of a burst.

If another attribute permutates against burstlength, it is considered to be completely permutated, if all of its values have occurred at all positions in all bursts of all specified burstlengths.

To ensure that all burstlengths are used, the generated master attribute page must be run with a block greater than the total of all listed burstlengths. Otherwise a burst could be interrupted before the end of the block.

NOTE A burstlength of one cannot be avoided totally.

Coverage

To get the coverage result, the master attributes are first permutated against required attributes 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 master attribute page size.

NOTE For an explanation of how the permutations are generated, refer to *"Generating Permutations" on page 178*.

Additionally, the master attributes of the `MA` group (address phase attributes) require permutation against `ML` (the burstlength `LAST`). The algorithm considers this when computing the coverage of the master attribute permutation.

Finally, the algorithm computes the number of data transfers required to achieve complete coverage by internally permutating the attribute 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 data transfers resulting from the number of groups and their lines to be permutated in the master attribute memory.

Test Area	Testing Time
T(MATTR)	Number of data transfers × Time-per-data-transfer

Block vs. Master Attribute Permutation

The master block parameters can be permutated against master attributes. Master attributes are permutated through their values when a compound block is executed repeatedly.

After the compound block has been completely executed with each master attribute group combination, all permutated block variation properties have been permutated against all permutated master attributes. The algorithm calculates how many block runs are needed to cover all required master attributes and their combinations and the amount of data to be transferred.

If the compound blocksize has been set to a power of 2, it is ensured that all permutated block variation properties have permutated against all permutated master attributes.

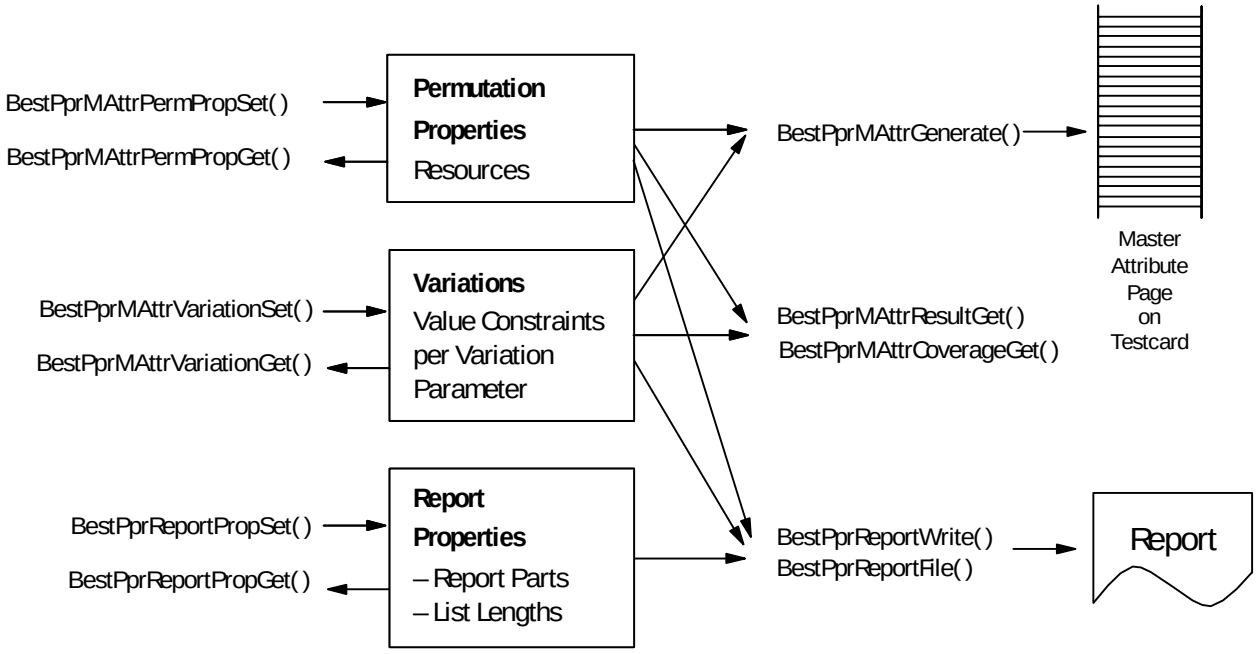
Testing Time

The testing time is determined by the testing time for the master block permutations and the number of master attribute permutations.

Test Area	Testing Time
T(ALL)	T(BLOCK) × Number of Attribute Permutations

Functions Overview

The following figure shows the master attribute and report functions used to prepare and to perform the permutation of the master attributes.



Programming Steps

Programming master attribute permutations requires the following steps:

1. Prepare a permutation by setting the permutation properties according to the resource requirements.

   Use *BestPprMAttrPermPropSet*.

   Master attribute permutation properties are:

   – the number of master attribute pages

   – the maximum page size

   – a starting point for the permutation algorithm

2. Define the lists of values for the variations and select the algorithm used to pick the values from this list.

   Use *BestPprMAttrVariationSet*.

   These lists specify values for the master attribute to be permutated according to the testing requirements.

3. If you want to check whether your test requirements are really suitable, request the test results or coverage.

   Use *BestPprMAttrResultGet* and *BestPprMAttrCoverageGet*.

4. Perform the permutation.

   There are two ways to do this:

   – A master attribute page can be generated and downloaded to the testcard, where it is used when running the master block page.

     Use *BestPprMAttrGenerate*.

   – The permutation results can be requested from the PCI Protocol Permutator and Randomizer software. The adjusted properties and the permutation results can then be written to a report file.

     Use *BestPprReportWrite* or *BestPprReportFile*.

NOTE The contents of the report file can be controlled with *BetPprReportPropSet* and *BetPprReportPropGet*.

Example

Program master attribute permutations as follows:

- Prepare the master attribute permutations by setting the following master attribute permutation properties:
  - Master attribute page 2 is to be used.
  - A maximum of 49 lines in the master attribute page may be allocated.

    This means that the Master Attribute Page Size (MAPS) must be less than 49 attribute lines.
- Define the following value lists for master attribute variations that should occur during the transfer, and select the algorithm that picks all values one after the other as listed in the value list.

Variations	Variation Parameter	Allowed Values
Master Attributes	last (burstlength) group: ML	4 8 32
	waits group: MD0	0 1 3 8
	steps group: MA1	0 7
	tryback group: MA1	true false

- Ensure that the following permutations are covered: .

Testing Area	Requirements	Tuple
MATTR	All burstlengths occur.	(burst)
	All numbers of wait cycles occur.	(wait)
	Steps 0 and 7 must occur.	(steps)
	Tryback occurs at least once or not at all.	(tryback)
	All counts of wait cycles meet all positions of all bursts of 4, 8, and 32 dwords.	(burst, wait)
	Steps = 0 must meet tryback = 1 at least once.	(steps, tryback)
ALL	All desired block permutations meet all desired master attribute permutations.	(BLOCK, MATTR)

Master Attribute Permutations

```
/* Set the master attribute permutation properties according to the
test design (testing area MATTR). */

/* Select master attribute page number 2. This page may contain up
to 49 attribute lines. */

    status = BestPprMAttrPermPropSet( handle,
                    BPPR_MA_PAGENUM, 2 ); CHECK;


    status = BestPprMAttrPermPropSet( handle,
                    BPPR_MA_PAGESIZEMAX, 49 ); CHECK;

/* Set the parameters for the master attribute variation. */

/* Set the required burstlengths of 4, 8 and 32.*/

    status = BestPprMAttrVariationSet( handle,
                    B_M_LAST,
                    "4, 8, 32",
                    BPPR_ALG_PERM ); CHECK;

/* Specify the variation parameters wait, steps, and tryback. */

    status = BestPprMAttrVariationSet( handle,
                    B_M_WAITS,
                    "0, 1, 3, 8",
                    BPPR_ALG_PERM ); CHECK;


    status = BestPprMAttrVariationSet( handle,
                    B_M_STEPS,
                    "0, 7",
                    BPPR_ALG_PERM ); CHECK;


    status = BestPprMAttrVariationSet( handle,
                    B_M_TRYBACK,
                    "true, false",
                    BPPR_ALG_PERM ); CHECK;

/* Note that you enter only your values of interest, but the
software inserts 0-values until the number of values is a prime. */

/* After setting up these parameters, generate the master attribute
page and download it to the testcard. Again, you may first check
and correct your test scenario. */

    status = BestPprMAttrGenerate( handle ); CHECK;
```

Programming Target Attribute Permutations

The target attribute permutations are used when the test scenario requires the Exerciser and Analyzer to be a PCI target. The target attributes can be constrained and permutated, similarly to the block variation or master attribute parameters described above. The results can be downloaded to a target attribute page on the testcard. The size of the target attribute page (TAPS) is selectable within the range limited by the testcard.

NOTE Because the bus control is on the side of the master, testing block permutations vs. target attribute permutations is not normally required.

Target Attributes

To achieve more sophisticated randomization opportunities, the target attributes are divided into groups that can be varied against each other. The following tables show which attributes are assigned to which group.

Group	Address Phase Attributes
TA0	64-bit Acknowledge System error signalling

Group	Data Phase Attributes
TD0	Waits
TD1	Termination, Parity and system errors

This assignment is fixed and cannot be programmed or otherwise changed.

Variation Parameters

A list of values and the algorithm to pick the values from the list can be specified for every target attribute.

Coverage

To get the coverage result, the target attributes are permutated as follows:

1. They are permutated against required attributes within their own group.
2. The resulting repetition length is raised up to the next higher unoccupied prime number.
3. The algorithm computes the number of data transfers required to achieve complete coverage by internally permutating the attribute groups against each other.

The group repetition lengths and the coverage information can be found in the report. Additionally they can be queried using a C-API function.

To guarantee that the target address phase attribute (attribute `APERR` in group `TA`) is actually executed, it must meet an address phase.

This is ensured if the exerciser is accessed in one of the following ways:

- Only a burstlength of 1 (single cycle) is used.

  In this case, each address phase is followed by one data phase.

- The address attribute is permutated against the target termination attribute TERM, containing a retry (1) or disconnect (2).

  In this case, the address attribute is followed by a retry or disconnect, and then meets the start of a new burst and thus a new address phase.

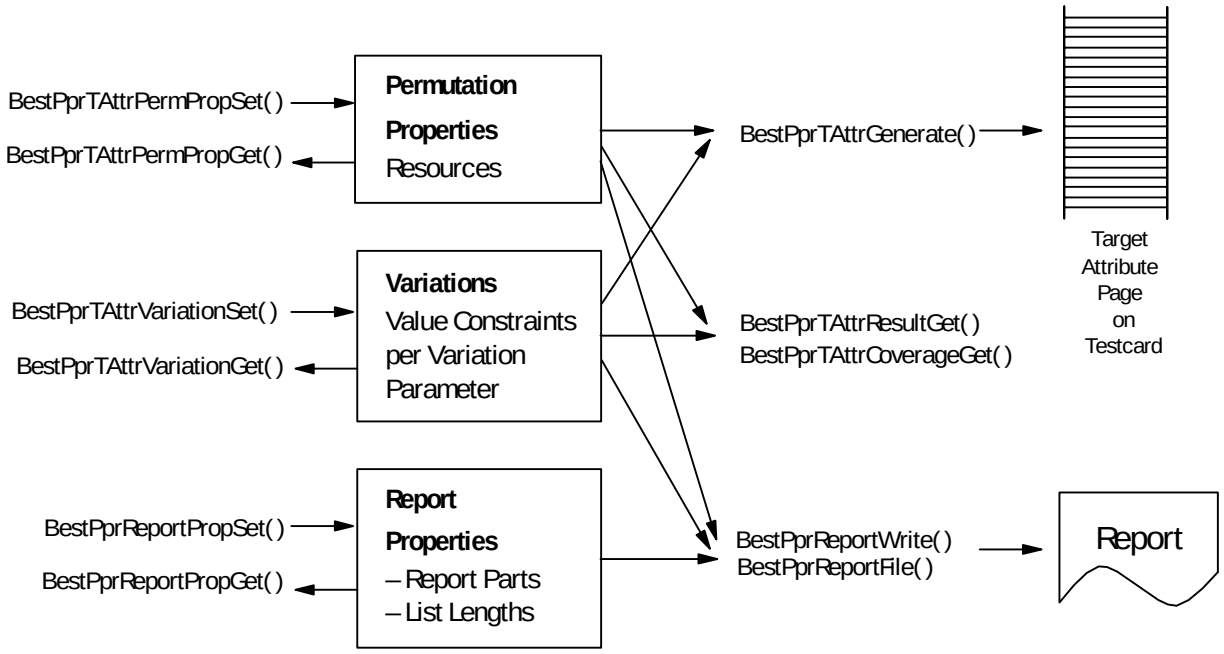
Testing Time

The testing time is determined by the number of PCI data transfers resulting from the number of groups and their lines to be permutated in the target attribute memory.

Test Area	Testing Time
T(TATTR)	Number of data transfers × Time-per-data-transfer

Functions Overview

The following figure shows the target attribute functions and the report functions used to prepare and perform the permutation of the target attributes.



Programming Steps

Programming target attribute permutations requires the following steps:

1. Prepare the permutation by setting the permutation properties according to the resource requirements.

   Use *BestPprTAttrPermPropSet*.

   Target attribute permutation properties are:

   – the number of the target attribute page

   – the maximum page size

   – a starting point for the permutation algorithm

2. Define the lists of values for the variations and select the algorithm used to pick the values from this list.

   Use *BestPprTAttrVariationSet*.

   These lists specify values for the target attributes to be permutated according to the testing requirements.

3. If you want to check whether your test requirements are really suitable, request the test results or coverage.

   Use *BestPprMAttrResultGet* and *BestPprMAttrCoverageGet*.

4 Perform the permutation.

There are two ways to do this:

– A target attribute page can be generated and downloaded to the testcard, where it determines the protocol level behavior of the testcard as a target.

Use *BestPprTAttrGenerate*.

– The permutation results can be requested from the PCI Protocol Permutator and Randomizer software. The adjusted properties and the permutation results can then be written to a report file.

Use *BestPprReportWrite* or *BestPprReportFile*.

NOTE The contents of the report file can be controlled with *BetPprReportPropSet* and *BetPprReportPropGet*.

Example

In general, the setup of the C code for programming target attribute permutations takes place in the same way as described for master attribute permutations.

Refer to *"Example" on page 203* for programming master attribute permutations.

Generating PPR Reports

The report contains the following information:

- Creation date and time
- Generic PPR properties
- Block permutations
- Master attribute permutations
- Target attribute permutations
- Information on master block vs. master attribute permutation
- Report properties
- C-API abbreviations
- Tables of:
  - Block permutation
  - Master attribute permutation
  - Target attribute permutation
- Hints and Warnings

  For more information, refer to *"Hints and Warnings in the Report String"* in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

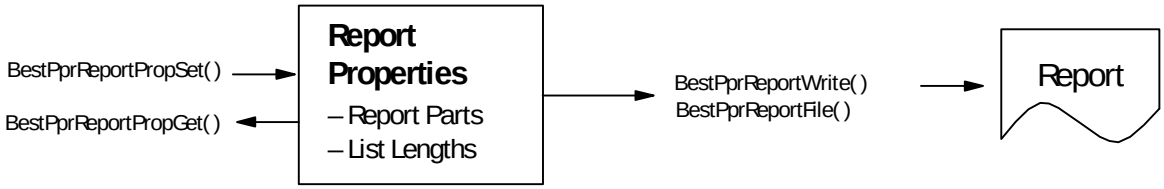
The contents of the report can be limited by setting report properties. For a list of constraints, refer to *"bppr_reportproptype"* in the *Agilent E2925B Opt. 320 C-API/PPR Reference*.

The report can be generated and written into a specified file.

The program generated for *"Example Test Design" on page 183* can be found in *"Analyzing the Report" on page 213*.

Functions Overview

The following figure shows the report functions used to prepare all kind 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 E2925B Opt. 320 C-API/PPR Reference*.

   Use *BetPprReportPropSet* and *BetPprReportPropGet*.

2. Request the permutation results from the PCI Protocol Permutator and Randomizer software and write all adjusted properties and the permutation results to a report file.

   Use *BestPprReportWrite* or *BestPprReportFile*.

Example

Task

Request a report that contains all test information except the target attributes. That means that the report of target attribute permutations and the report of the maximum number of target attribute page lines (starting with the first page line) must be skipped.

Setting Up the Report Properties

```
/* Set up report properties and print a report to a file. */

/* Skip the reports of the target attributes because the example
test scenario is set up for master operation. */

    status = BestPprReportPropSet (handle, BPPR_REP_TA, 0); CHECK;

    status = BestPprReportPropSet (handle, BPPR_REP_TACONTENT, 0);
   CHECK;

/* Generate the report and write it into the specified file. Note
that the report already contains the information on master block
vs. master attribute permutation being calculated by the algorithm.
The report generated by the program can be found in "Report
Listing" on page 232. */

    status = BestPprReportFile(handle, "report.txt"); CHECK;
```

Running the PPR Test

To run the PPR test, the PCI Permutator and Randomizer 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.

Programming Steps

Running a PPR Test requires the following steps:

1. Request the number of required block runs.

   Use *BestPprMAttrResultGet* or *BestPprTAttrResultGet* accordingly.

2. Run the test by running the block page as often as required.

3. Check for protocol errors occurred during the test by reading the bits in the testcard status register.

   Use *BestStatusRegGet*.

NOTE The test is observed by the functions of the protocol observer for protocol violations. The protocol observer is started automatically after power up and is per default set up to check all rules. For more information on the protocol observer, refer to *"Protocol Observer Programming" on page 47*.

4. If protocol errors have occurred, read out the observer'r result registers for information on the errors.

   Use *BestObsStatusGet*.

Example

Task Run the test specified under *"Example Test Design" on page 183* and check for protocol errors.

```
/* Before running the test, request the number of required block
runs. */

    status = BestPprMAttrResultGet( handle,
                     BPPR_MA_RUNS,
                     &blockruns );

/* Run the block page as often as required.*/

    printf ("Running master %u times\n", blockruns);
    for (count=0; count<blockruns; count++)
    {
       status = BestMasterBlockPageRun(handle, 1); CHECK;
       do

         {
          status = BestStatusRegGet(handle, &status_reg); CHECK;
         }
       while ( status_reg & 0x01);


       if (status_reg & 0x80)
       {
        printf ("Test failed, master abort has occured!\n");
        break;
       }
    }

/* If protocol errors have occurred, this is indicated by a bit in
the testcard's status register. Read out the observer's result
registers for information on the errors. */

    if (status_reg & 0x10)
    /* protocol error occured */
    {
     status = BestObsStatusGet (handle, B_OBS_ACCUERR,
     &status_reg); CHECK;


     printf("The following protocol errors have been detected:\n");
     for (errbit=1; errbit<=0x010000000; errbit >>=1)
     {
      if (status_reg & errbit)
      {
       status = BestObsErrStringGet (handle, errbit, &errtext);
       CHECK;
       printf ("%s\n", errtext);
   }
  }
 }
```

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 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" (`BPPR_REP_CAPI`), was active during program execution. Therefore, the C-API names of properties are given in brackets in each line. This information can be used to easily find the referring properties in your C program.

The reports of target attribute permutations and target attribute table are not considered in the example. These reports are similar to those of the master attribute permutations and the master attribute 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 232*.

Report Header

The report starts with a header containing the creation date and time, followed by general information.

```
Wed Mar 22 16:28:29 2000

    Agilent E2975A PCI Protocol Permutator & Randomizer SW
    ======================================================

    HW Type ..................................................... E2926A
    Connection Mode ............................................. ONLINE

    Generic Master Property (C-API)
       Master Attribute Page Mode ...... (B_MGEN_ATTRMODE) ... Sequential
```

General Properties

The general properties then follow, which are used to compute the testing times and a series of pseudo random numbers.

Refer to *"PPR Administration" on page 186*.

```
GENERIC PPR PROPERTIES
======================

   Bus Speed ..................... (BPPR_GEN_BUSSPEED) ..... 33 MHz
   Bus Width ..................... (BPPR_GEN_BUSWIDTH) ..... 32 bit
   Clocks per data transfer ...... (BPPR_GEN_XFERCLKS) ..... 10
   Seed .......................... (BPPR_GEN_SEED) ......... 0
```

Report of Block Permutations

Report of Master Block Permutation

This section of the report deals with the master block permutation. It shows which block and permutation properties are specified and which variations are constrained to which values.

```
BLOCK PERMUTATION
=================

Block Properties
   Bus address ..................... (B_BLK_BUSADDR) ........ 000b8000\h
   Transfer direction .............. (B_BLK_DIR) ............ Write
   Internal address ................ (B_BLK_INTADDR) ........ 0\h
   Compound block size ............. (B_BLK_NOFDWORDS) ...... 64
   Attribute page number ........... (B_BLK_ATTRPAGE) ....... 2
   Compare flag .................... (B_BLK_COMPFLAG) ....... 0
   Compare offset .................. (B_BLK_COMPOFFS) ....... 0\h

Permutation Properties
   Block page number ............... (BPPR_BLK_PAGENUM) ...... 1
   Max. master block page size ..... (BPPR_BLK_PAGESIZEMAX) .. 60
   Master block first permutation .. (BPPR_BLK_FIRSTPERM) .... 1
   Cacheline size in DWords ........ (BPPR_BLK_CACHELINE) .... 4
   Fill gaps ....................... (BPPR_BLK_FILLGAPS) ..... yes

Variation constraints (N = actual variation list length)

   Blocksize: .... perm, 3 values .. (BPPR_BLK_SIZE) ......... N=3
     "4,8,16"

   Byten: ........ fix ............. (BPPR_BLK_BYTEN) ........ N=1
     "all"

   Commands: ..... perm, 2 values .. (BPPR_BLK_CMDS) ......... N=2
     "Mem_write,Writeinvalidate"

   Alignment: .... perm, 5 values .. (BPPR_BLK_ALIGN) ........ N=5
     "(%16=0),(%16=4),(%16=8),(%16=12),(%32=0)"
```

Permutation Results The master block permutation section ends with the permutation results.

```
Permutation Results

   Last permutation ................ (BPPR_BLK_LASTPERM) ..... 30
   Actual page size ................ (BPPR_BLK_PAGESIZEACT) .. 26
   Estimated testtime .............. (BPPR_BLK_TIME) ......... 19.394 us

Coverage: covered, when R <= 30 (Last Permutation)
   (Blocksize) ..................................................... R=3   yes
   (Commands) ...................................................... R=30  yes
   (Alignment) ..................................................... R=5   yes
   (Blocksize, Commands) ........................................... R=30  yes
   (Blocksize, Alignment) .......................................... R=15  yes
   (Commands, Alignment) ........................................... R=30  yes
   (Blocksize, Commands, Alignment) ................................ R=30  yes

Total testtime T(BLOCK) ........................................... 19.394 us
   Gaps between blocks ............................................. 0 us
   Data transfer time .............................................. 19.394 us
```

Block Permutation Testing Considerations

To understand the results in the table above, there are certain restrictions to using the `MWI` command.

MWI Command Restrictions The use of the `MWI` command (PCI bus command 15) is restricted. The `MWI` command can only be used under the following circumstances:

- alignment is a multiple of the cacheline size
- all byte enables are active (that is, low)
- blocksize is a multiple of the cacheline size

The `MWI` command must not appear in other permutations. It will automatically be replaced by a non-disturbing memory command, so that the permutation of other variation parameters is not affected. This may result in duplicate entries in the permutation table. These duplicate entries will be deleted.

NOTE In this example test, `MWI` and burstlength are permutated against each other. This requires burstlength values greater or equal than cacheline size to prevent bursts from being interrupted within a cacheline. Therefore, in the example test the smallest burstlength to be permutated equals the cacheline size, which is assumed to be 4 dwords.

For the example test, this results in the block permutation table.

Block Permutation Table

The block permutation table shows the permutations to be executed. The tuple is given for each permutation. It is commented when the command had to be replaced. This may happen due to a combination with parameters. Duplicate tuples are also skipped.

```
Block Permutation Table
=======================

    PermNum       |     Size   Alignment   Byten    Command
------------------------------------------------------------------
        1         |       4     %16= 0     0000\b      7
        2         |       8     %16= 4     0000\b     15 (illegal-> 7)
        3         |      16     %16= 8     0000\b      7
        4         |       4     %16=12     0000\b     15 (illegal-> 7)
        5         |       8     %32= 0     0000\b      7
        6         |      16     %16= 0     0000\b     15
        7         |       4     %16= 4     0000\b      7
        8         |       8     %16= 8     0000\b     15 (illegal-> 7)
        9         |      16     %16=12     0000\b      7
       10         |       4     %32= 0     0000\b     15 (illegal-> 7)
       11         |       8     %16= 0     0000\b      7
       12         |      16     %16= 4     0000\b     15 (illegal-> 7)
       13         |       4     %16= 8     0000\b      7
       14         |       8     %16=12     0000\b     15 (illegal-> 7)
       15         |      16     %32= 0     0000\b      7
Skip, same as   1 |       4     %16= 0     0000\b     15 (illegal-> 7)
Skip, same as   2 |       8     %16= 4     0000\b      7
Skip, same as   3 |      16     %16= 8     0000\b     15 (illegal-> 7)
Skip, same as   4 |       4     %16=12     0000\b      7
Skip, same as   5 |       8     %32= 0     0000\b     15 (illegal-> 7)
       21         |      16     %16= 0     0000\b      7
Skip, same as   7 |       4     %16= 4     0000\b     15 (illegal-> 7)
Skip, same as   8 |       8     %16= 8     0000\b      7
Skip, same as   9 |      16     %16=12     0000\b     15 (illegal-> 7)
Skip, same as  10 |       4     %32= 0     0000\b      7
Skip, same as  11 |       8     %16= 0     0000\b     15 (illegal-> 7)
Skip, same as  12 |      16     %16= 4     0000\b      7
Skip, same as  13 |       4     %16= 8     0000\b     15 (illegal-> 7)
Skip, same as  14 |       8     %16=12     0000\b      7
       30         |      16     %32= 0     0000\b     15
```

The columns contain the permutation number, the blocksize transferred with the permutation, the alignment used, the byte enable (always low active) and the bus commands (7 or 15). The blocksizes of all valid permutation numbers add up to 172 bytes. This is less than the allocated compound blocksize of 64 dwords = 256 bytes. Thus there is a chance that all permutation numbers will fit into the compound blocksize.

NOTE This explains why the PCI Permutator and Randomizer 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, simply because they do not fit into this table. The length of this table may be diminished due to resource constraints.

Another reason why the PCI PPR software changes all parameters simultaneously is to achieve a good mix of test cases.

In the Block Permutation Table shown above, all block properties are permutated independently according to their value lists.

All variation list lengths (N) of the properties are prime numbers. If they were not, to calculate N, 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 178*).

- N for commands "15, 7" = 2
- N for blocksizes "4, 8, 16" = 3
- N for address alignments "%16=0, %16=4, %16=8, %16=12, %32=0" = 5.

The repetition length R is the length between two permutations with equivalent tuples. Thus, a block property has taken all values after R data transfers. For example: Three data transfers are necessary to test the 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 $5 \times 3 = 15$ data transfers. To combine all address alignments with all commands and blocksizes, you need $2 \times 3 \times 5 = 30$ 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)	5
(BLOCKSIZE)	3
(COMMAND)	2 × 5 × 3 = 30
(ALIGNMENT, BLOCKSIZE)	5 × 3 = 15
(ALIGNMENT, COMMAND)	5 × 3 × 2 = 30
(BLOCKSIZE, COMMAND)	5 × 3 × 2 = 30
(ALIGNMENT, BLOCKSIZE, COMMAND)	5 × 3 × 2 = 30
R(BLOCK)	Max(5, 3, 30, 15) = 30

For the commands, exceptions must be regarded. As described above, the `MWI` command may be changed by the software to meet the PCI specification. If this results in duplicate entries, these entries will be skipped, thus reducing the number of transfers.

If the number of permutations chosen is too small, it may occur that **no** `MWI` command is executed, although it is required in the variation parameter list. To avoid this situation, the `MWI` command must be permutated against all parameters that could prevent the occurrence of the `MWI` command. In the case of the example, these are both alignment and blocksize. Thus, the repetition length of the command tuple must be calculated as $2 \times 3 \times 5 = 30$.

NOTE These considerations are only true for algorithms that are not able to recognize the best command for a given tuple of variation parameters. If you chose the “best” algorithm, it automatically selects the best suitable command from the list. It then skips the permutations with other commands in order to decrease the repetition length.

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 64 dwords = 256 bytes.

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 by 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 Master Block Last Permutation (MBLP) can be queried, showing the number of the last permutation fitted into the compound block.

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 256 bytes. All 30 block permutations fit in (master block last permutation, MBLP = 30). Including the filled-in gaps, a total of 26 block transfers are used. This page size is below the limit defined by the maximum master block page size, which is 60 blocks in this example. The block fitting list shows schematically how the blocks are fitted into the compound block.

The report then produces the Block Fitting List, which shows the compound block resulting from the rearrangement of the example test.

```
Block Fitting List
==================

Actual Needed Size of Page: 26
First Permutation:          1


PermNum |  Start Addr    End Addr    Size   Alignment   Byten    Command
-------------------------------------------------------------------------
    5   |  0x000b8000   0x000b8007     8      %32= 0    0000\b      7
   13   |  0x000b8008   0x000b800b     4      %16= 8    0000\b      7
 fill   |  0x000b800c   0x000b800f     4
    1   |  0x000b8010   0x000b8013     4      %16= 0    0000\b      7
    7   |  0x000b8014   0x000b8017     4      %16= 4    0000\b      7
    3   |  0x000b8018   0x000b8027    16      %16= 8    0000\b      7
 fill   |  0x000b8028   0x000b802b     4
    9   |  0x000b802c   0x000b803b    16      %16=12    0000\b      7
 fill   |  0x000b803c   0x000b803f     4
   15   |  0x000b8040   0x000b804f    16      %32= 0    0000\b      7
   11   |  0x000b8050   0x000b8057     8      %16= 0    0000\b      7
 fill   |  0x000b8058   0x000b805f     8
   30   |  0x000b8060   0x000b806f    16      %32= 0    0000\b     15
 fill   |  0x000b8070   0x000b808f    32
   21   |  0x000b8090   0x000b809f    16      %16= 0    0000\b      7
 fill   |  0x000b80a0   0x000b80ab    12
   14   |  0x000b80ac   0x000b80b3     8      %16=12    0000\b      7
   12   |  0x000b80b4   0x000b80c3    16      %16= 4    0000\b      7
 fill   |  0x000b80c4   0x000b80cf    12
    6   |  0x000b80d0   0x000b80df    16      %16= 0    0000\b     15
   10   |  0x000b80e0   0x000b80e3     4      %32= 0    0000\b      7
 fill   |  0x000b80e4   0x000b80e7     4
    8   |  0x000b80e8   0x000b80ef     8      %16= 8    0000\b      7
 fill   |  0x000b80f0   0x000b80f3     4
    2   |  0x000b80f4   0x000b80fb     8      %16= 4    0000\b      7
    4   |  0x000b80fc   0x000b80ff     4      %16=12    0000\b      7

Last fit permutation: 30       Fit completely!
```

NOTE The length of the table printout may be restricted by the report properties. Here 30 lines are reported, which is the default value.

Summarizing the Results

The results for coverage and testing time of the master attribute permutations are described as follows.

Coverage In the example test, complete coverage is achieved. After 26 block permutations all required parameter combinations have occurred at least once.

The following table shows the repetition length for each tuple and whether it is covered after the Master Block Last Permutation (MBLP). Thus the testing goal for the BLOCK testing area is achieved.

Tuple	Repetition Length R	Coverage
(ALIGNMENT)	5	yes
(BLOCKSIZE)	3	yes
(COMMAND)	30	yes
(ALIGNMENT, BLOCKSIZE)	15	yes
(ALIGNMENT, COMMAND)	30	yes (was not a goal)
(BLOCKSIZE, COMMAND)	30	yes (was not a goal)
(ALIGNMENT, BLOCKSIZE, COMMAND)	30	yes (was not a goal)
R(BLOCK)	Max(5,3,30,15) = 30	yes (goal achieved)

Testing Time In the example test, the block test is performed after the 64 dwords (compound block size) have been transferred, that is after the 26 block transfers in the example. Assuming an average of 10 clock cycles for each of the 64 data transfers, less than 0.02 ms are needed for the data transfer (clock is 33 MHz, that is 30.3 ns per clock cycle).

Testing Time	Calculation	Value
Total T(BLOCK)	Data transfer time: 64 × 10 clocks × 30.3 ns	19.4 µs

Report of Master Attribute Permutation

This section reports the specified master attribute permutation parameters.

```
MASTER ATTRIBUTE PERMUTATION
============================

Permutation Properties
   Page number ..................... (BPPR_MA_PAGENUM) ....... 2
   Max. page size .................. (BPPR_MA_PAGESIZEMAX) ... 49
   First permutation ............... (BPPR_MA_FIRSTPERM) ..... 1
```

Variation Constraints

The following subsection shows the variation constraints for the master attribute permutation. It is reported which attributes of each group are permutated. Note that the repetition length of each group is raised up to the next prime.

```
Variation Constraints (R = Repetition Length)

  Group MA0 ...(requires permutation against ML)............. R=1
     DELAY:    fix =  "0"

  Group MA1 ...(requires permutation against ML)............. R=5
     STEPMODE: fix =  "no"
     STEPS:    permutated, 2 values = "0,7"
     TRYBACK:  permutated, 2 values = "yes,no"

  Group MA2 ...(requires permutation against ML)............. R=1
     RELREQ:   fix =  "on"
     REQ64:    fix =  "no"

  Group MA3 ...(requires permutation against ML)............. R=1
     RESUMEDELAY:fix =  "10"

  Group MA4 ...(requires permutation against ML)............. R=1
     APERR:    fix =  "no"
     AWRPAR:   fix =  "no"
     DACWP:    fix =  "no"
     DACPERR:  fix =  "no"
     AWP64:    fix =  "no"
     DACWP64:  fix =  "no"

  Group MD0 ................................................ R=7
     WAITS:    permutated, 4 values = "0,1,3,8"
     WAITMODE: fix =  "no"

  Group MD1 ................................................ R=1
     DPERR:    fix =  "no"
     DSERR:    fix =  "no"

  Group MD2 ................................................ R=1
     DWRPAR:   fix =  "no"
     DWP64:    fix =  "no"

  Group ML ................................................. R=47
     BURSTLEN: permutated, 3 values = "4,8,32"
```

In the example test design, three variation parameters have been specified for the master attribute testing area: burstlength, waits, steps and tryback (see *"Permutations to be Covered" on page 184*). The burstlength is of particular interest.

The **burstlength** is determined by an attribute called `LAST`. This attribute is set to 0 during a burst, and is set to 1 in the last data phase of a burst. In the attribute page, `LAST` is set to 1 if the permutation is the last data phase in the burst.

Required Blocksize

In the example test design, 3 bursts are required, one with a length of 1 dword, the second with 2 dwords and a third with 4 dwords. To have each burstlength occur at least once in the test, 4 + 8 + 32 = 44 data phases are required.

To make sure that bursts are performed with the lengths as intended, the master attribute page must be executed with a blocksize of at least the total of all burstlengths. Otherwise an intended burst would be interrupted by the end of the block. (Ideally, the used blocksize equals the total of all burstlengths. If it does not, a hint is given in the report.)

The value of 44 is increased to 47 to achieve a prime number as repetition length for coverage calculation.

Computing Repetition Lengths

The repetition lengths are computed by the algorithm as described in *"Generating Permutations" on page 178*.

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.

For the example test, the attributes of group `MA1` (`STEPS` and `TRYBACK`) have a group repetition length of 4. Therefore, the values 7 (`STEPS`) and 1 (`TRYBACK`) are filled in to obtain a repetition length of **5**.

The attribute `WAIT` has 4 values as well, which must be increased to the next unoccupied prime number. 5 is occupied by `MA1`, therefore, the values 0, 0, 1 are filled in to increase the repetition length to **7**.

The repetition length of group `ML` (`BURSTLENGTH`) is increased from 44 to **47**.

NOTE 2 is not used in this algorithm as a repetition length, even though it is a prime number. This guarantees that only attribute pages of odd lengths are generated (which is necessary for permutating attributes against blocks, see below).

The software first groups the attributes and performs a complete permutation within the group each attribute belongs to (MA1, MD0, ML). Afterwards each attribute is permutated through all of its possible values, similar to the permutation of block properties.

Master Attribute Permutation Table

After running the permutation algorithm, the attributes are permutated as shown in the following report section. Note that only permutated (not fixed) attribute parameters are listed..

```
Master Attribute Permutations
=============================
Actual Size of Page: 47
First Permutation:   1

--------------------
    |       B
  P |       U       T
  e |       R       R
  r |   W   S   S   Y
  m |   A   T   T   B
  N |   I   L   E   A
  u |   T   E   P   C
  m |   S   N   S   K
  --------------------
   1 |   0   0   0   1
   2 |   1   0   7   0
   3 |   3   0   7   1
   4 |   8   1   0   0
   5 |   0   0   7   1
   6 |   0   0   0   1
   7 |   1   0   7   0
   8 |   0   0   7   1
   9 |   1   0   0   0
  10 |   3   0   7   1

(Skipped)

  25 |   8   0   7   1
  26 |   0   0   0   1
  27 |   0   0   7   0
  28 |   1   0   7   1
  29 |   0   0   0   0
  30 |   1   0   7   1
Printout ends due to user setting (BPPR_REP_MACONTENT = 30).
------------------------------------------------
End of report.
```

The list shows the algorithm's operating principles. (It is limited to the first 30 permutations (of 1645 overall)).

NOTE Permutations 11 to 24 are skipped in this printout.

Only the varying parameters are included. The length of the table printout may be restricted by the *report properties*. In the following report section, 30 lines are reported, which is the default value.

Master Attribute Permutation Results

This subsection shows the master attribute permutation results and whether coverage is achieved under the given circumstances (that is, whether the repetition length of the tuple is below the specified page size).

```
Permutation Results
   Actual Page Size ................ (BPPR_MA_PAGESIZEACT) ... 47
   Last permutation number ......... (BPPR_MA_LASTPERM) ...... 1645
   Estimated testtime .............. (BPPR_MA_TIME) ......... 498.48 us
     testtime for  3-tuples......... (BPPR_MA_TUPLES_TIME) ... 498.48 us

Coverage: (n/a means not covered)
   (MA1) - requires ML ..................................... R= 235
   (MD0) ................................................... R= 7
   (ML) .................................................... R= 47
   (MA1, MD0) - requires ML ................................ R= 1645
   (MA1, ML) ............................................... R= 235
   (MD0, ML) ............................................... R= 329
   (MA1, MD0, ML) .......................................... R= 1645

   Max. covered single group ............................... 235
   Max. of covered pairs ................................... 1645
   Max. of covered 3-tuples ................................ 1645

Testtime total (MATTR)..................................... 498.48 us
Testtime (all  3-tuples, MATTR)............................ 498.48 us

Max. Data Transfer for all permutations (MATTR):........... 6.4258 Kbyte
Max. Data Transfer for all  3-tuples (MATTR):.............. 6.4258 Kbyte
Number of Block Page Runs needed for all permutations...... 26
Number of Block Page Runs needed for all  3-tuples......... 26
```

As for the block properties permutation, the algorithm used by the PPR software ensures that all attributes will have taken all their values at least once after `R` data phases, where `R` is the corresponding repetition length of the attribute tuples.

The algorithm permutates all tuples simultaneously, for example, all values are changed between two permutation steps (not only one value).

Calculating the Results

The results of the master attribute permutations as shown in the report section above are explained as follows.

Coverage The following table shows the maximum number of permutations required by the example test design for the `MATTR` testing area. This maximum of 47 entries of group `ML` fits into the master attribute page, which was assumed to have a size of 49 entries.

Note that the attributes of `MA1` (`STEPS` and `TRYBACK`) have effect only in address phases, and thus must be permutated against burstlength to make sure that it is covered. (This applies to all master attributes of group `MA` (master address phase attributes)).

Tuple	Attributes	Repetition Length R	Coverage
(ML)	BURST	47	yes
(MD0)	WAITS	7	yes
(MA1)	STEPS, TRYBACK (requires ML)	5 × 47 = 235	yes
(MA1, MD0)	WAITS, STEPS, TRYBACK (requires ML)	235 × 7 = 1645	yes
(MA1, ML)	STEPS, tryback, BURST	5 × 47 = 235	yes
(MD0, ML)	WAITS, BURST	7 × 47 = 329	yes
(MA1, MD0, ML)	STEPS, TRYBACK, WAITS, BURST	5 × 7 × 47 = 1645	yes

Testing Time The example master attribute test is completed after 1645 data transfers. Assuming an average of 10 clock cycles per data transfer at 33 MHz, this takes 498 µs.

Testing Time	Calculation	Value
T(MATTR) = Data transfer time	1645 transfers × 10 clocks × 30.3 ns	498 µs

In a 32-bit system, 1645 transfers means 6580 bytes (= 6.4258 Kbyte). In this example, the compound blocksize is set to 64 dwords, that is 256 bytes. Therefore, to transfer all required bytes, the block must be run 26 times at least (6580 / 256 = 25.7).

Hints Furthermore in the master attribute permutation section, the report has generated the following hint.

This hint informs you about the minimum block size of the master attribute page. If the master attribute page blocksize is smaller than this size, bursts will be aborted prematurely.

```
HINT:   This master attribute page should be called with
        a block size of at least 44 DWords to avoid shortened bursts.
```

Report of Master Block vs. Master Attribute Permutation

The following lines in the report result from the calculation of the complete testing time (test area ALL). This is the time it would take the test to run so that each block is permutated with each master attribute.

Refer to *"Block vs. Master Attribute Permutation" on page 201*.

```
BLOCK VS. MASTER ATTRIBUTE PERMUTATION
======================================

WARNING: Burstlengths cannot be guaranteed, because
         largest blocksize (4 DWords) is smaller than the
         sum of all burstlengths (44).

WARNING: MWI bursts must have sizes of multiple cachelines.
         Therefore make sure you use infinite burstlength
         when generating attribute pages by PPR,
         or set up your own attribute page so that a
         LAST bit will not interrupt transfer within a cacheline!

Testtime T(ALL) = T(BLOCK) * No. of Attribute permutations.. 31.903 ms
Testtime T(Tuples) = T(Block) * 1645 ..................... 31.903 ms
```

Calculating the Results

The results of the master attribute permutations as shown in the report section above are explained as follows.

First Warning The first warning is given because the largest blocksize of 4 dwords is smaller than the sum of all burstlengths of 44 dwords. Thus it cannot be guaranteed that each blocksize will be combined with each burstlength.

Second Warning The second warning appears because the `MWI` command is permutated against burstlength. This requires burstlength values greater than or equal to cacheline size to prevent transfers from being interrupted within a cacheline. The warning is a reminder for you to set up your test accordingly.

Testing Time For the testing area ALL, the block variation parameters are permutated against the master attributes. Therefore, the compound block is run repetitively while the master attribute group pages cycle through until any combination has occurred once.

To prevent runs of the compound block from using equivalent master attribute combinations, the compound blocksize (CBS) and the number of master attribute permutations) are not allowed to have a common divisor.

Therefore, CBS should have a power of 2 (here: 32 dwords). There is always an odd number of master attribute permutations, because it is the result of multiplied prime numbers above 2.

In the example test, all permutations are covered after the compound block has been executed 1645 times.

Testing Time	Calculation	Value
T(ALL)	T(BLOCK) × Number of Attribute permutations = 19.4 µs × 1645	31.9 ms

Report of Report Properties

This section shows the report properties specified when this report was created. The report properties can be set by the command *BestPprReportPropSet*.

```
REPORT
======

Properties
   Print general properties ........ (BPPR_REP_GEN) .......... yes
   Print block properties .......... (BPPR_REP_BLK) .......... yes
   Print master attribute properties (BPPR_REP_MA) ........... yes
   Print target attribute properties (BPPR_REP_TA) ........... no
   Print report properties ......... (BPPR_REP_REPORT) ....... yes
   Print block page lines........... (BPPR_REP_BLOCKCONTENT) . 30
   Print master attr. page lines ... (BPPR_REP_MACONTENT) .... 30
   Print target attr. page lines ... (BPPR_REP_TACONTENT) .... 0
   Max. order of tuple listed ...... (BPPR_REP_ORDER_TUPLES) . 3
   Print C-language symbols ........ (BPPR_REP_CAPI) ......... yes
```

Block Page Contents

This section shows how the block page is programmed by the Permutator and Randomizer Software after use of the function *BestPprBlockGenerate*.

```
Block Page Contents
===================

Size of Page: 26

  BusAddr    Command   Byten   NOfDWords   IntAddr
  ------------------------------------------------
  0x000b8010     7      0000\b       1      0x00010
  0x000b80f4     7      0000\b       2      0x000f4
  0x000b8018     7      0000\b       4      0x00018
  0x000b80fc     7      0000\b       1      0x000fc
  0x000b8000     7      0000\b       2      0x00000
  0x000b80d0    15      0000\b       4      0x000d0
  0x000b8014     7      0000\b       1      0x00014
  0x000b80e8     7      0000\b       2      0x000e8
  0x000b802c     7      0000\b       4      0x0002c
  0x000b80e0     7      0000\b       1      0x000e0
  0x000b8050     7      0000\b       2      0x00050
  0x000b80b4     7      0000\b       4      0x000b4
  0x000b8008     7      0000\b       1      0x00008
  0x000b80ac     7      0000\b       2      0x000ac
  0x000b8040     7      0000\b       4      0x00040
  0x000b8090     7      0000\b       4      0x00090
  0x000b8060    15      0000\b       4      0x00060
  0x000b800c     7      0000\b       1      0x0000c
  0x000b8028     7      0000\b       1      0x00028
  0x000b803c     7      0000\b       1      0x0003c
  0x000b8058     7      0000\b       2      0x00058
  0x000b8070     7      0000\b       8      0x00070
  0x000b80a0     7      0000\b       3      0x000a0
  0x000b80c4     7      0000\b       3      0x000c4
  0x000b80e4     7      0000\b       1      0x000e4
  0x000b80f0     7      0000\b       1      0x000f0
```

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 burstlengths 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 page automatically in order to fill gaps, or to increase the number of bursts to a prime number.

- Vary either burstlength or blocksize.
- Vary only attributes of interest.
- The algorithm chooses the best suitable PCI bus command.

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 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 Permutator and Randomizer software or parts of it by means of the `...Init` functions.

After initialization, the default values can be set with the `...DefaultSet` functions.

Requesting Results

In the example, the results are written to a report file only. The application programming interface provides further possibilities to request the following:

- particular permutation results (with `...ResultGet` functions)
- repetition length and coverage of tuples (`...CoverageGet`)
- specified properties (`...PropsGet`)
- specified variation parameters (`...VariationGet`)

This information is also written to the report (unless it is suppressed by the report properties settings).

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.

More Exhaustive Test

The example test uses greatly reduced hardware resources to keep the example output short (report, tables). A more exhaustive test would use nearly all the hardware resources of the exerciser testcard:

- The master block page size can hold a maximum number of 256 blocks (Master Block Page Size MBPS = 256).
- For each of the nine master attribute groups, 250 entries can be programmed to be permutated against each other. This results in up to 250^9 ($= 3.8 \times 10^{21}$) attribute permutations (and therefore in a virtually infinite test duration).
- The typical compound blocksize (CBS) is the half of the testcard's data memory (64 KBytes, that is 16K dwords), so that the other half of the memory can be used for a read/write compare reference data. Refer to "*Data Compare Unit*" in the *Agilent E2925B Opt. 300 PCI Exerciser User's Guide*.

Uncovered Permutations

If the coverage of permutated tuples is not achieved (although it is required), the report will contain a hint listing possible reasons. 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 Permutator and Randomizer 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.
- `...PAGENUM`, which allows assignment of different pages in the internal memory.

These properties can be used to fill the different pages with attributes or blocks. The algorithm can be set to continue where the previous invocation stopped.

Reproducing Bus Errors

If the bus hangs during execution of a block page, the function *BestMasterBlockStatusRead* can be used to determine the last block that was executed successfully.

This function returns the number of the block and the position in the master attribute page used when starting execution of the block. This information can then be used to restart the permutation algorithm with new start values.

Report Listing

This section contains the report of the example described in *"Example Test Design" on page 183*.

```
Wed Mar 22 16:28:29 2000

     Agilent E2975A PCI Protocol Permutator & Randomizer SW
     ==============================================

     HW Type ..................................................... E2926A
     Connection Mode ............................................. ONLINE

     Generic Master Property (C-API)
        Master Attribute Page Mode ...... (B_MGEN_ATTRMODE) ... Sequential



     GENERIC PPR PROPERTIES
     ======================

        Bus Speed ....................... (BPPR_GEN_BUSSPEED) ..... 33 MHz
        Bus Width ....................... (BPPR_GEN_BUSWIDTH) ..... 32 bit
        Clocks per data transfer ........ (BPPR_GEN_XFERCLKS) ..... 10
        Seed ............................ (BPPR_GEN_SEED) ......... 0
```

```
BLOCK PERMUTATION
=================

Block Properties
   Bus address ..................... (B_BLK_BUSADDR) ........ 000b8000\h
   Transfer direction .............. (B_BLK_DIR) ............ Write
   Internal address ................ (B_BLK_INTADDR) ........ 0\h
   Compound block size ............. (B_BLK_NOFDWORDS) ...... 64
   Attribute page number ........... (B_BLK_ATTRPAGE) ....... 2
   Compare flag .................... (B_BLK_COMPFLAG) ....... 0
   Compare offset .................. (B_BLK_COMPOFFS) ....... 0\h

Permutation Properties
   Block page number ............... (BPPR_BLK_PAGENUM) ...... 1
   Max. master block page size ..... (BPPR_BLK_PAGESIZEMAX) .. 60
   Master block first permutation .. (BPPR_BLK_FIRSTPERM) .... 1
   Cacheline size in DWords ........ (BPPR_BLK_CACHELINE) .... 4
   Fill gaps ....................... (BPPR_BLK_FILLGAPS) ..... yes

Variation constraints (N = actual variation list length)

   Blocksize: .... perm, 3 values .. (BPPR_BLK_SIZE) ......... N=3
     "4,8,16"

   Byten: ........ fix ............. (BPPR_BLK_BYTEN) ........ N=1
     "all"

   Commands: ..... perm, 2 values .. (BPPR_BLK_CMDS) ......... N=2
     "Mem_write,Writeinvalidate"

   Alignment: .... perm, 5 values .. (BPPR_BLK_ALIGN) ........ N=5
     "(%16=0),(%16=4),(%16=8),(%16=12),(%32=0)"

Permutation Results

   Last permutation ................ (BPPR_BLK_LASTPERM) ..... 30
   Actual page size ................ (BPPR_BLK_PAGESIZEACT) .. 26
   Estimated testtime .............. (BPPR_BLK_TIME) ......... 19.394 us

Coverage: covered, when R <= 30 (Last Permutation)
   (Blocksize) ..................................................... R=3   yes
   (Commands) ...................................................... R=30  yes
   (Alignment) ..................................................... R=5   yes
   (Blocksize, Commands) ........................................... R=30  yes
   (Blocksize, Alignment) .......................................... R=15  yes
   (Commands, Alignment) ........................................... R=30  yes
   (Blocksize, Commands, Alignment) ................................ R=30  yes

Total testtime T(BLOCK) ............................................ 19.394 us
   Gaps between blocks ............................................. 0 us
   Data transfer time .............................................. 19.394 us
```

```
MASTER ATTRIBUTE PERMUTATION
============================

Permutation Properties
   Page number ..................... (BPPR_MA_PAGENUM) ....... 2
   Max. page size .................. (BPPR_MA_PAGESIZEMAX) ... 49
   First permutation ............... (BPPR_MA_FIRSTPERM) ..... 1

Variation Constraints (R = Repetition Length)

   Group MA0 ...(requires permutation against ML)..... R=1
      DELAY:    fix =  "0"

   Group MA1 ...(requires permutation against ML)..... R=5
      STEPMODE: fix =  "no"
      STEPS:    permutated, 2 values = "0,7"
      TRYBACK:  permutated, 2 values = "yes,no"

   Group MA2 ...(requires permutation against ML)..... R=1
      RELREQ:   fix =  "on"
      REQ64:    fix =  "no"

   Group MA3 ...(requires permutation against ML)..... R=1
      RESUMEDELAY:fix =  "10"

   Group MA4 ...(requires permutation against ML)..... R=1
      APERR:    fix =  "no"
      AWRPAR:   fix =  "no"
      DACWP:    fix =  "no"
      DACPERR:  fix =  "no"
      AWP64:    fix =  "no"
      DACWP64:  fix =  "no"

   Group MD0 ....................................... R=7
      WAITS:    permutated, 4 values = "0,1,3,8"
      WAITMODE: fix =  "no"

   Group MD1 ....................................... R=1
      DPERR:    fix =  "no"
      DSERR:    fix =  "no"

   Group MD2 ....................................... R=1
      DWRPAR:   fix =  "no"
      DWP64:    fix =  "no"

   Group ML ........................................ R=47
      BURSTLEN: permutated, 3 values = "4,8,32"


Permutation Results
   Actual Page Size ................ (BPPR_MA_PAGESIZEACT) ... 47
   Last permutation number ......... (BPPR_MA_LASTPERM) ...... 1645
   Estimated testtime .............. (BPPR_MA_TIME) .......... 498.48 us
     testtime for  3-tuples......... (BPPR_MA_TUPLES_TIME) ... 498.48 us
```

```
Coverage: (n/a means not covered)
   (MA1) - requires ML ..................................... R= 235
   (MD0) ..................................................... R= 7
   (ML) ...................................................... R= 47
   (MA1, MD0) - requires ML ................................ R= 1645
   (MA1, ML) ................................................. R= 235
   (MD0, ML) ................................................. R= 329
   (MA1, MD0, ML) ........................................... R= 1645

   Max. covered single group ................................ 235
   Max. of covered pairs .................................... 1645
   Max. of covered 3-tuples ................................. 1645

Testtime total (MATTR)..................................... 498.48 us
Testtime (all  3-tuples, MATTR)............................ 498.48 us

Max. Data Transfer for all permutations (MATTR):........... 6.4258 Kbyte
Max. Data Transfer for all  3-tuples (MATTR):.............. 6.4258 Kbyte
Number of Block Page Runs needed for all permutations...... 26
Number of Block Page Runs needed for all  3-tuples......... 26


HINT:  This master attribute page should be called with
       a block size of at least 44 DWords to avoid shortened bursts.
```

```
BLOCK VS. MASTER ATTRIBUTE PERMUTATION
======================================

WARNING: Burstlengths cannot be guaranteed, because
         largest blocksize (4 DWords) is smaller than the
         sum of all burstlengths (44).

WARNING: MWI bursts must have sizes of multiple cachelines.
         Therefore make sure you use infinite burstlength
         when generating attribute pages by PPR,
         or set up your own attribute page so that a
         LAST bit will not interrupt transfer within a cacheline!

Testtime T(ALL) = T(BLOCK) * No. of Attribute permutations.. 31.903 ms
Testtime T(Tuples) = T(Block) * 1645 ..................... 31.903 ms



REPORT
======

Properties
   Print general properties ........ (BPPR_REP_GEN) .......... yes
   Print block properties .......... (BPPR_REP_BLK) .......... yes
   Print master attribute properties (BPPR_REP_MA) ........... yes
   Print target attribute properties (BPPR_REP_TA) ........... no
   Print report properties ......... (BPPR_REP_REPORT) ....... yes
   Print block page lines........... (BPPR_REP_BLOCKCONTENT) . 30
   Print master attr. page lines ... (BPPR_REP_MACONTENT) .... 30
   Print target attr. page lines ... (BPPR_REP_TACONTENT) .... 0
   Max. order of tuple listed ...... (BPPR_REP_ORDER_TUPLES) . 3
   Print C-language symbols ........ (BPPR_REP_CAPI) ......... yes
```

```
Block Permutation Table
=======================

    PermNum       |     Size   Alignment   Byten    Command
------------------------------------------------------------------------
        1         |        4     %16= 0     0000\b      7
        2         |        8     %16= 4     0000\b     15 (illegal-> 7)
        3         |       16     %16= 8     0000\b      7
        4         |        4     %16=12     0000\b     15 (illegal-> 7)
        5         |        8     %32= 0     0000\b      7
        6         |       16     %16= 0     0000\b     15
        7         |        4     %16= 4     0000\b      7
        8         |        8     %16= 8     0000\b     15 (illegal-> 7)
        9         |       16     %16=12     0000\b      7
       10         |        4     %32= 0     0000\b     15 (illegal-> 7)
       11         |        8     %16= 0     0000\b      7
       12         |       16     %16= 4     0000\b     15 (illegal-> 7)
       13         |        4     %16= 8     0000\b      7
       14         |        8     %16=12     0000\b     15 (illegal-> 7)
       15         |       16     %32= 0     0000\b      7
Skip, same as   1 |        4     %16= 0     0000\b     15 (illegal-> 7)
Skip, same as   2 |        8     %16= 4     0000\b      7
Skip, same as   3 |       16     %16= 8     0000\b     15 (illegal-> 7)
Skip, same as   4 |        4     %16=12     0000\b      7
Skip, same as   5 |        8     %32= 0     0000\b     15 (illegal-> 7)
       21         |       16     %16= 0     0000\b      7
Skip, same as   7 |        4     %16= 4     0000\b     15 (illegal-> 7)
Skip, same as   8 |        8     %16= 8     0000\b      7
Skip, same as   9 |       16     %16=12     0000\b     15 (illegal-> 7)
Skip, same as  10 |        4     %32= 0     0000\b      7
Skip, same as  11 |        8     %16= 0     0000\b     15 (illegal-> 7)
Skip, same as  12 |       16     %16= 4     0000\b      7
Skip, same as  13 |        4     %16= 8     0000\b     15 (illegal-> 7)
Skip, same as  14 |        8     %16=12     0000\b      7
       30         |       16     %32= 0     0000\b     15
```

```
Block Fitting List
==================

Actual Needed Size of Page: 26
First Permutation:          1


PermNum |  Start Addr    End Addr    Size   Alignment   Byten    Command
-------------------------------------------------------------------------
    5   |  0x000b8000   0x000b8007     8      %32= 0    0000\b      7
   13   |  0x000b8008   0x000b800b     4      %16= 8    0000\b      7
 fill   |  0x000b800c   0x000b800f     4
    1   |  0x000b8010   0x000b8013     4      %16= 0    0000\b      7
    7   |  0x000b8014   0x000b8017     4      %16= 4    0000\b      7
    3   |  0x000b8018   0x000b8027    16      %16= 8    0000\b      7
 fill   |  0x000b8028   0x000b802b     4
    9   |  0x000b802c   0x000b803b    16      %16=12    0000\b      7
 fill   |  0x000b803c   0x000b803f     4
   15   |  0x000b8040   0x000b804f    16      %32= 0    0000\b      7
   11   |  0x000b8050   0x000b8057     8      %16= 0    0000\b      7
 fill   |  0x000b8058   0x000b805f     8
   30   |  0x000b8060   0x000b806f    16      %32= 0    0000\b     15
 fill   |  0x000b8070   0x000b808f    32
   21   |  0x000b8090   0x000b809f    16      %16= 0    0000\b      7
 fill   |  0x000b80a0   0x000b80ab    12
   14   |  0x000b80ac   0x000b80b3     8      %16=12    0000\b      7
   12   |  0x000b80b4   0x000b80c3    16      %16= 4    0000\b      7
 fill   |  0x000b80c4   0x000b80cf    12
    6   |  0x000b80d0   0x000b80df    16      %16= 0    0000\b     15
   10   |  0x000b80e0   0x000b80e3     4      %32= 0    0000\b      7
 fill   |  0x000b80e4   0x000b80e7     4
    8   |  0x000b80e8   0x000b80ef     8      %16= 8    0000\b      7
 fill   |  0x000b80f0   0x000b80f3     4
    2   |  0x000b80f4   0x000b80fb     8      %16= 4    0000\b      7
    4   |  0x000b80fc   0x000b80ff     4      %16=12    0000\b      7

Last fit permutation: 30       Fit completely!
```

```
Block Page Contents
===================

Size of Page: 26

   BusAddr    Command   Byten   NOfDWords   IntAddr
  ------------------------------------------------
  0x000b8010     7      0000\b       1      0x00010
  0x000b80f4     7      0000\b       2      0x000f4
  0x000b8018     7      0000\b       4      0x00018
  0x000b80fc     7      0000\b       1      0x000fc
  0x000b8000     7      0000\b       2      0x00000
  0x000b80d0    15      0000\b       4      0x000d0
  0x000b8014     7      0000\b       1      0x00014
  0x000b80e8     7      0000\b       2      0x000e8
  0x000b802c     7      0000\b       4      0x0002c
  0x000b80e0     7      0000\b       1      0x000e0
  0x000b8050     7      0000\b       2      0x00050
  0x000b80b4     7      0000\b       4      0x000b4
  0x000b8008     7      0000\b       1      0x00008
  0x000b80ac     7      0000\b       2      0x000ac
  0x000b8040     7      0000\b       4      0x00040
  0x000b8090     7      0000\b       4      0x00090
  0x000b8060    15      0000\b       4      0x00060
  0x000b800c     7      0000\b       1      0x0000c
  0x000b8028     7      0000\b       1      0x00028
  0x000b803c     7      0000\b       1      0x0003c
  0x000b8058     7      0000\b       2      0x00058
  0x000b8070     7      0000\b       8      0x00070
  0x000b80a0     7      0000\b       3      0x000a0
  0x000b80c4     7      0000\b       3      0x000c4
  0x000b80e4     7      0000\b       1      0x000e4
  0x000b80f0     7      0000\b       1      0x000f0
```

```
Master Attribute Permutations
=============================

Actual Size of Page: 47
First Permutation:   1

---------------------
     |       B
  P  |       U       T
  e  |       R       R
  r  |   W   S   S   Y
  m  |   A   T   T   B
  N  |   I   L   E   A
  u  |   T   E   P   C
  m  |   S   N   S   K
  --------------------
   1 |   0   0   0   1
   2 |   1   0   7   0
   3 |   3   0   7   1
   4 |   8   1   0   0
   5 |   0   0   7   1
   6 |   0   0   0   1
   7 |   1   0   7   0
   8 |   0   0   7   1
   9 |   1   0   0   0
  10 |   3   0   7   1
  11 |   8   0   0   1
  12 |   0   1   7   0
  13 |   0   0   7   1
  14 |   1   0   0   0
  15 |   0   0   7   1
  16 |   1   0   0   1
  17 |   3   0   7   0
  18 |   8   0   7   1
  19 |   0   0   0   0
  20 |   0   0   7   1
  21 |   1   0   0   1
  22 |   0   0   7   0
  23 |   1   0   7   1
  24 |   3   0   0   0
  25 |   8   0   7   1
  26 |   0   0   0   1
  27 |   0   0   7   0
  28 |   1   0   7   1
  29 |   0   0   0   0
  30 |   1   0   7   1
Printout ends due to user setting (BPPR_REP_MACONTENT = 30).



------------------------------------------------
End of report.
```

Index

N

O

P

R

S

T

U

V

W

Publication Number: 5988-4894EN

Agilent Technologies