HP E2928A

66 MHz PCI Exerciser and Analyzer

In Stock

Used and in Excellent Condition

Open Web Page

https://www.artisantg.com/54709-2

HP E2920 Computer Verification Tools, PCI Series

User's Guide

HP E2925A

PCI Bus Exerciser and Analyzer

Document Revision 5.1 for software revision 4.00.00
July 1998

Notice

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this manual.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the rights in Technical Data and Computer Software Clause at DFARS 252.227.7013.
Hewlett-Packard Co., 3000 Hanover Street, Palo Alto, CA 94304.

Microsoft, Visual C++, Windows and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark licensed exclusively through X/Open Ltd.

Hewlett-Packard GmbH
Boeblingen Verification Solutions
Herrenbergerstrasse 130
71034 Boeblingen
Germany

About this manual

This manual describes the HP E2925A Bus Exerciser and Analyzer. The operating principles of the card and its control interfaces are described. This manual also describes the C application programming interface (C-API) and command line interface (CLI). Examples are provided for the C-API and CLI.

Product options and add-ons are described in separate documents.

This manual is organized as follows:

Chapter 1, Product Overview

This chapter describes the product's key features, and an overall product structure (product numbers and options)

Chapter 2, Using the Graphical User Interface

This chapter introduces the graphical user interface.

Chapter 3, Using the Command Line Interface

This chapter introduces the command line interface.

Chapter 4, Using the C-API

This chapter demonstrates the use of C-API

Chapter 5, PCI Exerciser Overview

This chapter describes the operation and use of the exerciser.

Chapter 6, PCI Analyzer Overview

This chapter describes the operation and use of the analyzer, including the protocol observer, protocol rules, trace memory, triggering and storage qualification.

Chapter 7, Programming Reference

This chapter describes each C-API function and the associated data

types and constants.

Chapter 8, Programming Quick Reference

This chapter provides a summary of the available C-API functions and return codes.

Chapter 9, Control and Programming Interfaces

This chapter describes the interfaces used to control the HP E2925A card. The RS232, EPP and PCI control interfaces are described.

Chapter 10, DUT and Instrument Interfaces

This chapter describes the mailbox registers, CPU port, Static I/O port logic analyzer adapter and external trigger I/O.

Chapter 11, Miscellaneous Interfaces

This chapter describes LEDs, hex. display and external power connector.

Chapter 1 Product Overview

Chapter 2 Using the Graphical User Interface

Chapter 3 Using the Command Line Interface

Chapter 4 Using the C-API

Chapter 5 PCI Exerciser Overview

Chapter 6 PCI Analyzer Overview

Chapter 7 Programming Reference

Chapter 8 Programming Quick Reference

Chapter 9 Control and Programming Interfaces

Chapter 10 DUT and Instrument Interfaces

Chapter 11 Miscellaneous Interfaces

Chapter 1 Product Overview

This chapter gives the product key features, and an overall product structure (product numbers and options).

This chapter contains the following sections:

What you can do with the HP E2925A

The HP E2925A can be used in the following ways:

- Generate PCI traffic and emulate a PCI target

  The HP E2925A provides master and target state-machines and master and target attribute memory which define the attributes of each address/data phase.

  See *Using the built-in test functions. on page 30*, *Creating Master Transactions on page 32*, *Creating Master Transactions on page 46*, *Master Operation on page 112*, *Master Programming on page 116*, *Creating Target Transactions on page 54*, *Programming the configuration space of the E2925A on page 84*, *Target Programming on page 122* and *Configuration Space on page 128*.

- Analyze PCI traffic and trigger on protocol violations.

  See *Triggering the Analyzer Trace Memory on page 60*, *Programming the Protocol Observer on page 64*, *Analyzer Overview on page 140* and *Protocol Observer on page 141*.

- Inspect and modify memory, I/O or configuration space of any PCI device.

  See *Using the Host access functions on page 67* and *Host to/from PCI System Memory on page 135*.

- Download test code to device under test (DUT).

  See *Using the onboard expansion ROM on page 87* and *Using the Host access functions on page 67*.

- Provide a link between an external host and the DUT.

  See *Communicating with the DUT via the mailbox registers on page 79* and .

- Set up and control devices in a DUT or validation system.

  See *Using the CPU port on page 70* and *Using the Static I/O port on page 74*

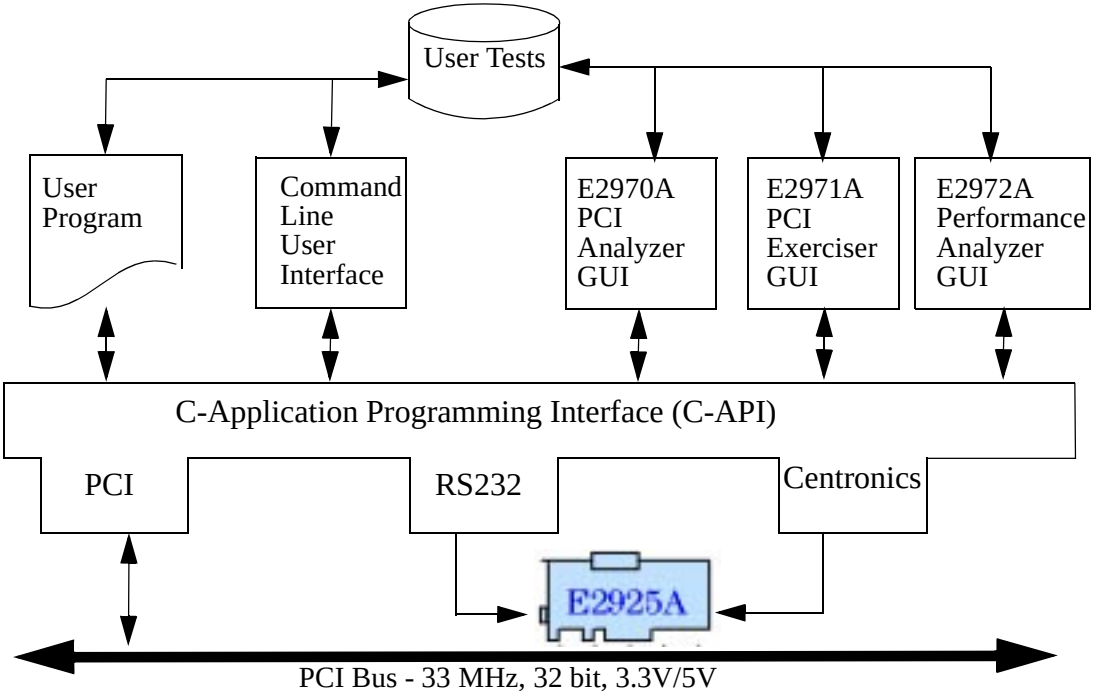
HP E2925A Key Features

- 32 Bit, 33MHz PCI device with programmable configuration space.
- Complete C-API programming interface
- Graphical User Interface control
- Command Line Interface control
- Programmable master protocol behavior (e.g. bursts and waits)
- Concurrent PCI target with programmable behavior (e.g. termination and waits)
- Programmable exception control (e.g. wrong parity,PERR#,SERR#)
- 64 kByte Expansion EEPROM
- 32k * 32 Bit onboard memory, accessible as PCI memory or I/O space. For simulation of larger memory this can be mirrored.
- Five programmable target decoders:
    - memory from 4k byte to 16M byte
    - I/O or memory from 16 byte to 64k byte.
    - 32 byte I/O or Configuration access to programming registers
    - Expansion EEPROM (256k byte)
    - Configuration Space decoder
- 32k state deep onboard logic analyzer
- All 32 bit PCI signals and statemachine outputs are stored in the onboard analyzer trace memory. Tracememory is controlled by a programmable trigger generator and sample qualifier, providing PCI logic analyzer capabilities.
- PCI protocol observer with 25 protocol rules according to spec 2.1
- All PCI signals are connected to expansion connectors for a piggyback board for performance measurements.
- Direct clock (no PLL), enables the system to operate in an environment with switched clock (power management mode).
- Pattern comparator and delay counter for conditional and timed master start

HP E2920 Series Product Structure

The HP E2920 Series product has a modular structure, which allows you to combine the hardware with different user interface and software blocks. This can be configured depending on your specific application.

Figure 1 Product Overview



The HP E2925A Product and Product Options

The HP E2925A consists of :

- 32 Bit, 33 MHz PCI Exerciser and Analyzer card
- C-Application Programming Interface (C-API)
- Command Line Interface (CLI) for Windows 95/NT
- Software ID module to enable additional software product options
- Serial Cable
- Installation Guide

The following HP E2925A options are available:

- External Power Supply (#001)
- Fast Host Interface (#002)
- HP Logic Analyzer Adapter (#003)
- Generic Logic Analyzer Adapter (#004)
- Performance/Deep Trace memory board (#100) (See performance option board User's Guide)

The following additional products may be used in conjunction with the HP E2925A

- E2970A PCI Analyzer Graphical User Interface
- E2971A PCI Exerciser Graphical User Interface
- E2972A PCI Performance Analyzer Graphical User Interface
- E2975A Protocol Permutator and Randomizer

C-Application Programming Interface (C-API) The C-API is a library of C functions that provides a complete programming interface to the card. The compiled executable may run on the system under test communicating through the PCI bus, or run on an external controller communicating through RS-232 or Bi-directional Centronics.

Command Line Interface (CLI) The CLI provides the programmer with an interactive means of programming and controlling the card from the host. CLI commands can also be entered into a batch file to create a test. Nearly all C-API functions are available to the CLI.

External Power Supply (Option 001) This option provides an external power supply to prevent the card from drawing power from its slot.

Fast Host Interface (Option 002) This option includes an ISA Bi-directional Centronics interface card and cable for applications requiring high data upload/download throughput.

HP Logic Analyzer Adapter (Option 003) This option provides a daughter card with all onboard PCI analyzer signals and appropriate terminations for direct connection to an external HP logic analyzer.

Generic Logic Analyzer Adapter (Option 004) This option provides a daughter card with all onboard PCI analyzer signals and appropriate terminations for direct connection to an external HP logic analyzer.

Chapter 2 Using the Graphical User Interface

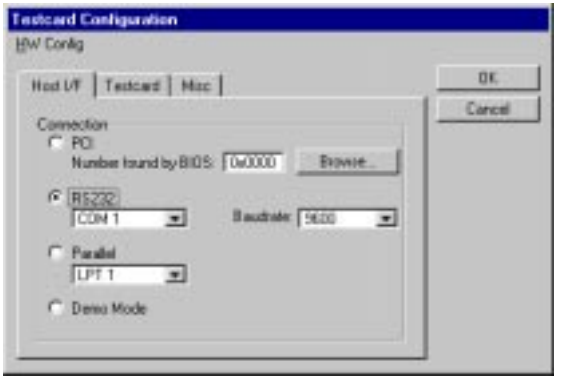
This chapter describes how to use the GUI framework included with the HP E2925A.

Setting up a connection in the GUI

Software installation is described in the Installation Guide.
The standard GUI also provides some system setup windows. Before you can use the GUI to control the E2925A card, you must first configure the interface port you will use to communicate with the card.

1 From the Main Window Setup menu, select Testcard Configuration:

2 From the Testcard Configuration window, select the Host Interface port you are using:

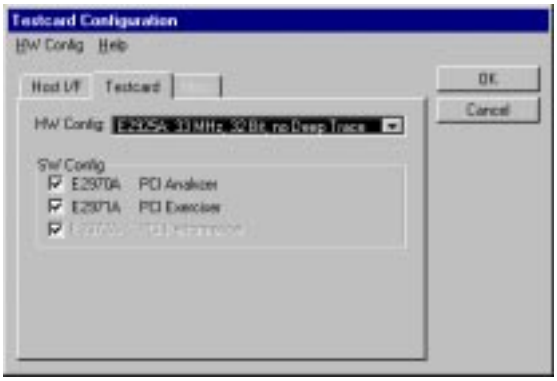


PCI
The device under test (DUT) is also the host computer and communicates using the PCI bus. A device number may be entered directly or may be selected from the available devices by pressing the Browse button.
RS232
The host and DUT are physically or logically separated and communicate over the serial interface.
Parallel
Host is an external controller communicating using the Fast Host Interface Card.

Selecting card type and product options

A range of hardware and software options is available for the E2925A. These may be enabled from the Testcard Configuration window.

Software upgrades are described in the Installation Guide.



E2925A, 33 MHz, 32 Bit, Deep Trace must be selected to use the HP E2972A Performance Analyzer GUI.

Chapter 3 Using the Command Line Interface

This chapter describes how to use the command line interface (CLI).

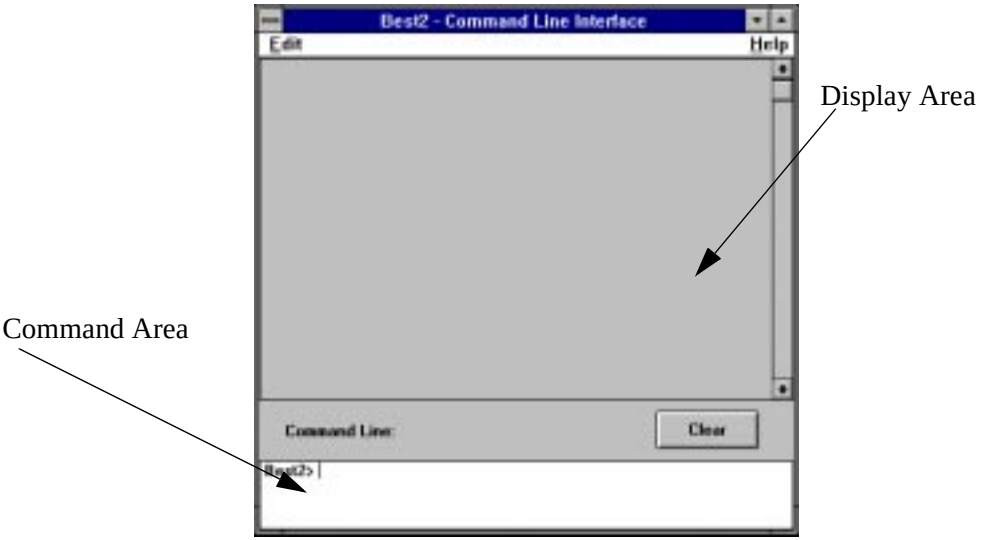
This chapter contains the following sections:

Overview of the CLI

The CLI provides an interface to the C API which can be used for simple interactive testing or to execute batch files containing more complex test sequences.

Command Line Interface Window

This window is divided into two areas:



Display Area Each command typed in the command area or executed in a batch file is displayed here. Error messages and return values of functions are also displayed in this window.

Command Area Commands can be typed in this area and are executed when Return is pressed. Several commands separated by semicolons can be typed on one line. The last 4 commands can be recalled by using the up-cursor and down-cursor keys. A recalled command can be edited and executed again by pressing Return.

Commands can be executed from a batch file by typing

do *path*

where *path* is the path and filename. If no path is specified the interpreter searches in the current directory. A batch file may be generated:

- using a text editor
- using command logging
- by copying typed commands and pasting them into the batch file

Successfully executed commands can be logged to a log file.

Commands

There are two types of commands:

- C-API function calls
- Run batch file

C-API function calls These are equivalent to the corresponding C-API functions, but are not case sensitive. There is an abbreviation for each command and its parameters.

The general syntax of a function call is:

function-name parameter = value ...

Data passed to a function through a pointer can be typed:

pointername = &{ value_1, value_2, ..., value_n}

or redirected from a file:

pointername < filename

Data returned from a function through a pointer is returned in the display window by omitting the pointer parameter in the command, or may be redirected to a file:

pointername > filename

Run batch file The *do* keyword followed by the name and path of the batch file runs the commands in the file.

Using the built-in test functions.

Using the built-in test functions involves the following steps:

1 Define the test properties

2 Define the master generic properties

3 Initiate the test run

4 Upload the test results

Test Properties

The following test properties are used to control the built-in test functions:

- Bandwidth
- Blocklength
- Data pattern (random, fix, toggle)
- Compare data read with data written
- Protocol variation (light, medium, hard)
- Start address
- Number of bytes transferred

Master Generic Properties can be used to control how data is transferred during the test run.

Initiate the Test Run

The following test commands are available:

- Protocol Error Detect
  Set up the on-board logic analyzer to capture protocol violations.
- Traffic Make
  Perform writes to the card's own target, thus consuming bandwidth.
- Write-Read/Write-Read-Compare
  Perform writes and reads to a PCI memory resource specified by the start address. The function can optionally compare the data read with the data written.
- BlockMove
  Copies a block of data from one PCI address to another.
- Read

Performs reads on the PCI bus.

Upload the Test Results

The test results can be uploaded using the testrdump function. This saves the analyzer and observer status, including the trace memory contents.

```
testrdump file="c:\temp\xx.rpt"
```

Summary and Example

The test functions are used as follows:

1. Call mstop to ensure that the master is not running.
2. Call testprpset once for each test property you want to change.
3. Call mgrpset once for each generic property you want to change.
4. Call testrun once to run the test with the specified command.
5. Call testrdump to get the results of the test run.

The following example performs writes and reads to system memory resource:

```
mstop

testprpdefset
testprpset prop=start val=b8000\h
testprpset prop=nob val=160
testprpset prop=dpat val=dptoggle
testprpset prop=prot val=hard
testprpset prop=comp val=1
mgprpset prop=repmode val=infinite
testrun cmd=writeread
sregget
testrdump file="c:\temp\xx.rpt"
```

Creating Master Transactions

Creating master transactions from BEST involves the following steps:

1. Define 1 or more block transfers
2. Define protocol behavior attributes for the block transfer
3. Define the generic run properties
4. Initiate the block run

Block Transfer

Block transfers are the basic programming constructs for generating master transactions. For more information on block transfers see “Master Block Transfer” on page 114.

To program a master block transfer

1. Call mbpginit once with the block page number you want to program. This page number is referenced by the master block run function.
2. Call mbprpdefset once, to set the preparation register to the defaults.
3. Call mbprpset once for each attribute you want to change in the preparation register
4. Call mbprog once to program the master block memory with the content of the preparation register. This automatically increments the programming pointer to the master block.
5. Repeat steps 3 and 4 for each line (address/data phase) you want to program.

For example, the following extract programs 1 master block transfer read of 28\h dwords, from video memory address 0xb8e60, to internal memory address 0, using the protocol attributes in attribute memory page 0x0 (default protocol attributes):

```
mbpginit page=01
mbprpdefset

  mbprpset prop=bad val=b8e60\h
  mbprpset prop=iad val=0
  mbprpset prop=cmd val=memread
  mbprpset prop=nod val=0x28
  mbprpset prop=apage val=0x0
  mbprog
```

Programming Protocol Attributes

For each data phase you can:

- program the amount of waits
- signal PERR#
- signal SERR#
- invert parity
- force release of REQ#
- specify if a phase is the last phase of a burst.

Programmable address phase attributes are SERR#, address stepping and exclusive access. If the loop attribute is set the current page is repeated. For more information on attributes and their default values see “b_mattrproptype” on page 177.

Protocol attributes are stored in an attribute memory which is divided into pages. All attributes in page 0 are set to their default values and cannot be overwritten.

After power up, all attributes are programmed with their default values, with the exception of loop bit which is set to one.

To program an attribute page:

1 Call mapginit once, with the attribute page number you want to program. This page number is referenced by the master block.

2 Call maprpdefset once, to set the preparation register to defaults (no loop)

3 Call maprpset once for each attribute you want to change in the preparation register

4 Call maphprog once to program the attribute memory line with the content of the preparation register. This automatically increments the programming pointer to the next attribute memory line.

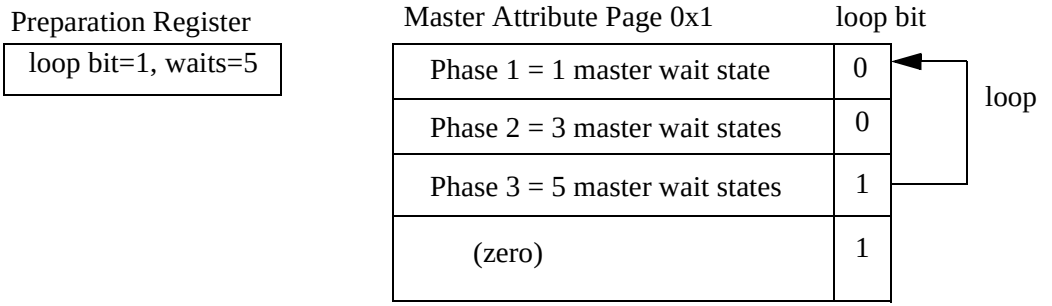
5 Repeat steps 3 and 4 for each line (address/data phase) you want to program. Remember to set the loop bit in the last line, in order to loop the structure.

For example, the following extract programs 3 attribute phases, with increasing values of wait states :

```
mapginit page=01
maprpdefset
maprpset prop=w val=1
maphprog
```

```
maprpset prop=w val=3
maphprog

maprpset prop=w val=5
maprpset prop=loop val=1
maphprog
```

This leaves the following in the preparation register and in master attribute memory



This attribute page may then be used as follows:

```
mbprpset prop=apage val=1
mbprog
```

Generic Run Properties

Generic run properties define how block transfers are to be executed.

The following example programs an immediate, single master transfer. Although the mode is immediate, it is started only after the run function is called.

```
mgprpdefset
```

The following example programs the master block to start after a trigger pattern is seen on the bus, and then runs indefinitely.

```
mgprpdefset
mgprpset prop=runmode val=wondelay
mcspset patt= “!FRAME & AD32==b8xxx\h”
mgprpset prop=repmode val=infinite
```

Block Run

To start the transactions on the bus either *BestMasterBlockRun ( ) on page 171*, or *BestMaster-BlockPageRun ( ) on page 172* must be called. The block run function runs the master block that is currently defined in the preparation register. The page run function runs the specified page, which may contain one or more programmed blocks.

The following function call runs block page 1:

```
mbpgrun page=1
```

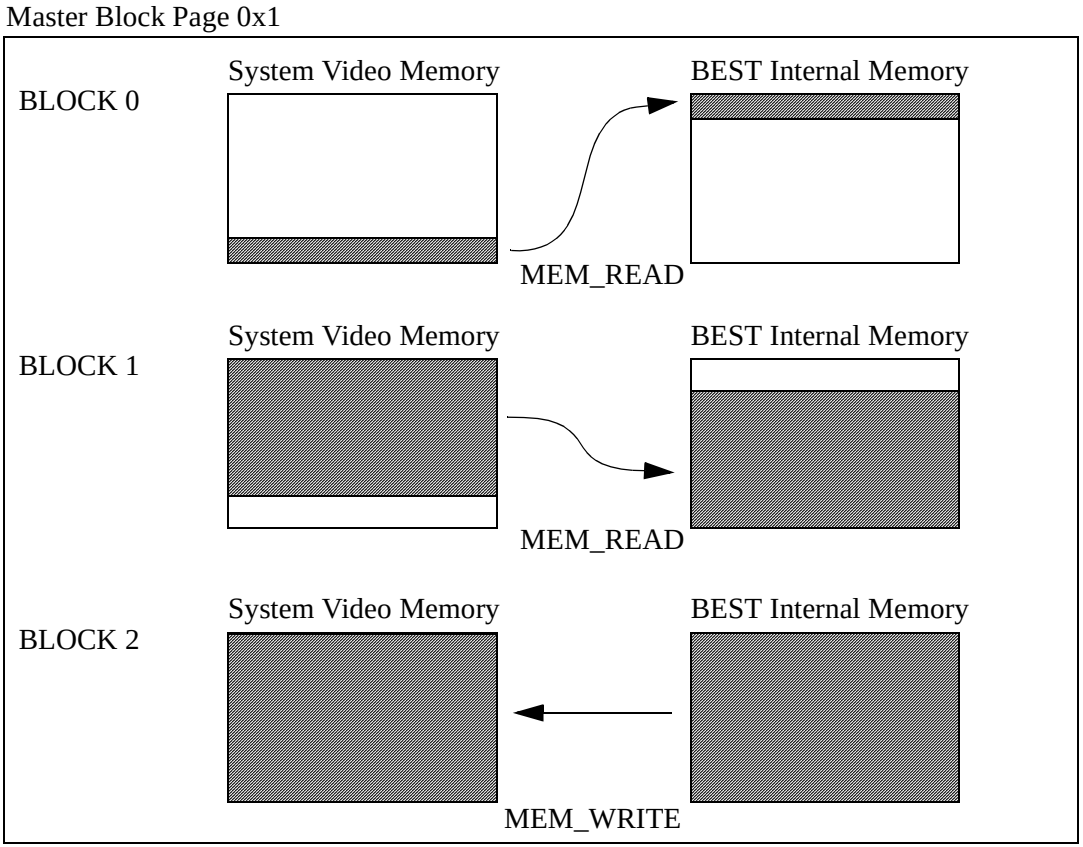
Complete Master Programming Example

The following example scrolls the screen, until mstop is typed. It can be restarted by typing

```
mbpgrun page=01
```

The example uses 5 protocol attribute phases using 1, 3, 5, 7 and 9 wait states repeatedly

It uses 3 master block transfers which are executed consecutively (chained).



Block 0, performs a memory read of the last system video memory line to internal address 0.

Block 1, reads the rest of the video memory, into internal memory below the first line.

Block 3 writes the new internal memory video image, to system video memory.

Master Transaction Example

```
mapginit page=01
maprpdefset

  maprpset prop=w val=1
  maphprog
  maprpset prop=w val=3
  maphprog
  maprpset prop=w val=5
  maphprog
  maprpset prop=w val=7
  maphprog
  maprpset prop=w val=9
  maprpset prop=loop val=1
  maphprog

mbpginit page=01
  mbprpdefset
  mbprpset prop=bad val=b8e60\h
  mbprpset prop=iad val=00
  mbprpset prop=cmd val=mem_read
  mbprpset prop=nod val=28\h
  mbprpset prop=apage val=1
  mbprog

  mbprpdefset
  mbprpset prop=bad val=b8000\h
  mbprpset prop=iad val=a0\h
  mbprpset prop=cmd val=mem_read
  mbprpset prop=nod val=3c0\h
  mbprpset prop=apage val=1
  mbprog

  mbprpdefset
  mbprpset prop=bad val=b8000\h
  mbprpset prop=cmd val=mem_write
  mbprpset prop=nod val=3e8\h
  mbprpset prop=apage val=1
  mbprog

mgprpdefset
mgprpset prop=runmode val = wondelay
mgprpset prop=repmode val=infinite
mbpgrun page=01
```

Chapter 4 Using the C-API

This chapter describes how to use the C application programming interface (C-API). This chapter contains the following sections:

Compiling C-API applications

If you are using Microsoft Visual C++ Developer Studio, you will find a complete project template under the samples\capi in the installation directory. The source code for all examples in this chapter is also available in the samples\capi directory.

Compiling and linking a C-API application involves:

- Adding <installation dir>\include\ to the include path
- Linking the application against <installation dir>\capi32.lib (Windows 95/NT) or a versiopn of the C-API library which you have compiled.

NOTE: If you are running Windows 95/NT, you must include the 'bin' subdirectory of the installation directory in your PATH to run the examples (CAPINT.DLL and CAPI95.DLL are located there).

Compiling the C-API library

Source code for the C-API is provided in the <installation dir>\sources\[capidos | capi95 | capint] directory.

Compiling the C-API library involves:

- Adding <installation dir>\include\ and <installation dir>\sources\[capidos | capi95 | capint] to the include path
- Defining WIN32 if compiling for Windows NT
- Linking with advapi32.lib and user32.lib if compiling for Windows NT/95

NOTE: If compiling the C-API library for an OS other than Windows NT, ensure that WIN32 is not defined.

- Linking against advapi32.lib and user32.lib (Windows NT/95).

Opening and Closing the Connection to the Card

The first set of function calls in a C-API application establish a communication channel with the card. The last function call should close this communication channel.

Sequence of C-API Function Calls

1. If using the PCI Bus as the controlling interface port then:

   Call "BestDevIdentifierGet ( )" on page 149., which returns the device number of the card (as used by the PCI BIOS). This will be used in BestOpen().

2. Initialize Internal Structures and Variables for Interface Port

   Function call "BestOpen ( )" on page 150.

3. Set Baud Rate

   If using the RS232 serial interface "BestRS232BaudRateSet ( )" on page 152.

4. Establish Connection to Card

   Function call "BestConnect ( )" on page 153.

5. Application Code

   This is where you enter your application code.

6. Disconnect, so port no longer has exclusive access, (other ports may now connect)

   Function call "BestDisconnect ( )" on page 154.

7. Close the session and deallocate memory

   Function call "BestClose ( )" on page 155.

Handling Errors

Nearly all C-API functions return an error code. The error code returned by a function call can be processed and the corresponding error string printed using function *BestErrorStringGet ( ) on page 270*. The recommended method of processing function call errors is shown below:

```
b_errtype err;
b_handletype handle;

err=BestOpen(&handle,B_PORT_RS232,B_PORT_COM1);
if (err != B_E_OK)
    {printf ("%s\n", BestErrorStringGet(err)); return MyErrorCcde;}
```

In addition to using returned error values, the status register may be used to obtain the current status of the card. (See *BestStatusRegGet ( ) on page 252* and *BestStatusRegClear ( ) on page 254*).

```
b_errtype err;
b_handletype handle;
b_int32 status;

err=BestOpen(&handle,B_PORT_RS232,B_PORT_COM1);
err=BestStatusRegGet(handle, &status);
if(status & (1<<7)) {
  printf("Block aborted!\n");
  BestStatusRegClear(handle, 1<<7);
}
```

BestStatusRegClear clears the specified bit in the status registers.

RS232 Example

This example opens a connection to the card using the serial port and sets the baud rate to 57600 bps.

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

#define CHECK if (status != B_E_OK) {printf ("%s\n", \
  BestErrorStringGet(status));return -1;}


int main ( )
{

b_errtype status;
b_handletype handle;

b_charptrtype product_number;

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

/*Establish Connection to card*/
BestConnect ( handle );CHECK

/*Set baud rate to 57600*/
BestRS232BaudRateSet(handle,B_BD_57600);CHECK
```

```
/* Application program goes in here, for example:*/
/* Read product number from card firmware:*/
BestVersionGet (handle, B_VER_PRODUCT, &product_number);CHECK

/* disconnect from the current port*/
BestDisconnect (handle);CHECK

/* close the session and deallocate memory*/
BestClose(handle);CHECK
}
```

EPP Example

The following example opens a connection to the card using the parallel port 2

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

#define CHECK if (status != B_E_OK) {printf ("%s\n", \
  BestErrorStringGet(status));return -1;}


int main ( )
{

b_errtype status;
b_handletype handle;

b_charptrtype product_number;

/*Initialize port internal structs and variables*/
BestOpen(&handle,B_PORT_PARALLEL,B_PORT_LPT2);CHECK

/*Establish Connection to card*/
BestConnect ( handle );CHECK

/* Application program goes in here, for example:*/
/* Read product number from card firmware:*/
BestVersionGet (handle, B_VER_PRODUCT, &product_number);CHECK

/* disconnect from the current port*/
BestDisconnect (handle);CHECK

/* close the session and deallocate memory*/
BestClose(handle);CHECK
```

```
}
```

PCI Example

The following example opens a connection to two cards using the PCI interface. The *BestDevIdentifierGet ( )* call can be used to obtain the device identifier or, if the slotnumber of the card is known, this can be used in the device identifier.

The format of the device identifier is as follows

bits 15:8 bus number

bits 7:3 slot number

bits 2:0 function number

The third parameter is an index used to identify the card when multiple cards are used (assuming they have the same vendor id and device id).

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

#define CHECK if (status != B_E_OK) {printf (“%s\n”, \
  BestErrorStringGet(status));return -1;}

int main ( )
{

b_errtype status;
b_handletype handle1, handle2;
b_int32 devid;

b_charptrtype product_number;

/*Get device number of first card
The index number can be used to distinguish between
multiple cards*/
BestDevIdentifierGet(0x103C, 0x2925, 0, &devid);CHECK
/*Initialize port internal structs and variables*/
BestOpen(&handle1, B_PORT_PCI, devid);CHECK

/*Get device number of second card (number=1)*/
BestDevIdentifierGet(0x103C, 0x2925, 1, &devid);CHECK
```

```
BestOpen(&handle2, B_PORT_PCI, devid);CHECK

/*Establish Connection to cards*/
BestConnect ( handle1 );CHECK
BestConnect ( handle2 );CHECK

/* Application program goes in here, for example:*/
/* Read product number from card firmware:*/
BestVersionGet (handle, B_VER_PRODUCT, &product_number);

/* disconnect from the current port*/
BestDisconnect (handle1);CHECK
BestDisconnect (handle2);CHECK

/* close the session and deallocate memory*/
BestClose(handle1);CHECK
BestClose(handle2);CHECK
}
```

Creating Master Transactions

Creating master transactions from BEST involves the following steps:

1. Define 1 or more block transfers
2. Define protocol behavior attributes for the block transfer
3. Define the generic run properties
4. Initiate the block run

Block Transfer

Block transfers are the basic programming constructs for generating master transactions. For more information on block transfers see “Master Block Transfer” on page 114.

The programming mechanism is very similar to programming attribute memory, however the memories are completely independent.

To program a master block transfer

1. Call *BestMasterBlockPageInit ( )* once with the block page number you want to program. This page number is referenced by the master block run function.
2. Call *BestMasterBlockPropDefaultSet ( )* once, to set the preparation register to the defaults.
3. Call *BestMasterBlockPropSet ( )* once for each attribute you want to change in the preparation register
4. Call *BestMasterBlockProg ( )* once to program the master block memory with the content of the preparation register. This automatically increments the programming pointer to the master block.
5. Repeat steps 3 and 4 for each line (address/data phase) you want to program.

For example, the following source code extract programs 1 master block transfer read of 28\h dwords, from video memory address 0xb8e60, to internal memory address 0, using the protocol attributes in attribute memory page 0x0 (default protocol attributes): (Note: no error handling is shown):

```
BestMasterBlockPageInit(handle,0x01);
BestMasterBlockPropDefaultSet(handle);

  BestMasterBlockPropSet(handle,B_BLK_BUSADDR,0xb8e60);
  BestMasterBlockPropSet(handle,B_BLK_INTADDR,0x00)
```

```
BestMasterBlockPropSet(handle,B_BLK_BUSCMD,B_CMD_MEM_READ);
BestMasterBlockPropSet(handle,B_BLK_NOFDWORDS,0x28);
BestMasterBlockPropSet(handle,B_BLK_ATTRPAGE,0x0);
BestMasterBlockProg(handle);
```

Programming Protocol Attributes

Protocol attributes for a block transfer are defined within master attribute memory. This memory consists of 8k lines or phases which can be divided up into a maximum of 256 pages. For more info on memory organization see “Master Programming” on page 116.

Protocol attributes are programmed on a per phase basis, where each attribute memory line corresponds to an address or data phase. Each attribute memory line contains a set of attributes for address phases and a set of attributes for data phases. If the current phase is an address phase, then the address attributes are used. If the current phase is a data phase then the address attributes are ignored, and the data phase attributes are used. For each data phase you can program the amount of waits, signal PERR#, signal SERR#, invert parity, force release of REQ#, specify if the phase is the last phase of a burst. Programmable address phase attributes are SERR#, address stepping and exclusive access. If the DOLOOP attribute is set the current page is repeated. For more information on master protocol attributes and their default values see “b_mattrproptype” on page 177.

Attribute memory page zero cannot be overwritten, and contains all default attributes values (zero), but with the loop bit set.

After power up, all attribute memories are programmed with zero, with the exception of DOLOOP bit which is set to one. This means any attribute page can be referenced from the master block command without disastrous consequences.

To program an attribute page:

1. Call *BestMasterAttrPageInit ( )* once, with the attribute page number you want to program. This page number is referenced by the master block.
2. Call *BestMasterAttrPropDefaultSet ( )* once, to set the preparation register to defaults (no loop)
3. Call *BestMasterAttrPropSet ( )* once for each attribute you want to change in the preparation register
4. Call *BestMasterAttrPhaseProg ( )* once to program the attribute memory line with the content of the preparation register. This automatically increments the programming pointer to the next attribute memory line.
5. Repeat steps 3 and 4 for each line (address/data phase) you want to program. Remember to set the

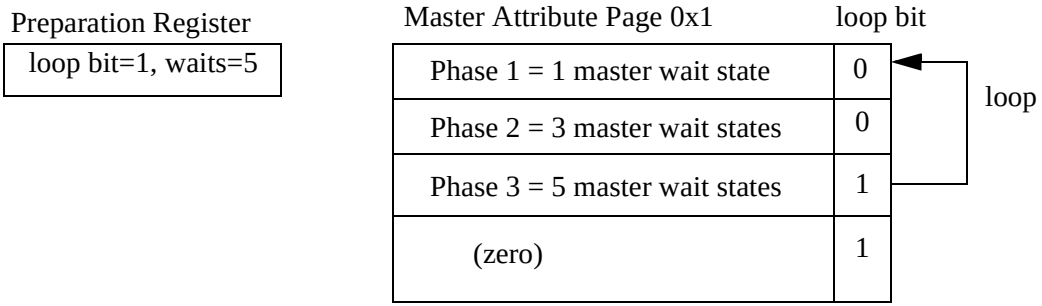
DOLOOP bit in the last line, in order to loop the structure.

For example, the following source code extract programs 3 attribute phases, with increasing amount of wait states (no error handling shown):

```
BestMasterAttrPageInit(handle,0x01);
BestMasterAttrPropDefaultSet(handle);

/* First Address or Data Phase */
BestMasterAttrPropSet(handle,B_M_WAITS,1);
BestMasterAttrPhaseProg(handle);

/* Second Address or Data Phase */
BestMasterAttrPropSet(handle,B_M_WAITS,3);
BestMasterAttrPhaseProg(handle);

/* Third Address or Data Phase, and goto first */
BestMasterAttrPropSet(handle,B_M_WAITS,5);
BestMasterAttrPropSet(handle,B_M_DOLOOP,1);
BestMasterAttrPhaseProg(handle);
```

BestMasterAttrPtrSet ( ) can be used to move the master attribute programming pointer to any offset relative to the start of a page and *BestMasterAttrPhaseRead ( )* can be used to read the attributes of the current phase.

This leaves the following in the preparation register and in master attribute memory



This attribute page may then be used as follows:

```
BestMasterBlockPropSet(handle,B_BLK_ATTRPAGE,0x1);
BestMasterBlockProg(handle);
```

Generic Run Properties

Generic run properties define how block transfers are to be executed. They define the run mode (immediate, with trigger pattern delay and/or CPU timer delay, and the delay values), the repeat mode (run the block once or indefinitely), and the latency timer.

Run properties are not programmed using a preparation register.

The following example programs an immediate, single master transfer. Although the mode is immediate, it is started only after the run function is called.

```
BestMasterGenPropDefaultSet(handle);
```

The following example programs the master block to start after a trigger pattern is seen on the bus, and then run indefinitely.

```
BestMasterGenPropDefaultSet(handle);

BestMasterGenPropSet(handle, B_MGEN_RUNMODE, B_RUNMODE_WONDELAY);

BestMasterCondStartPattSet(handle,
   “b_state=3\\h & CBE3_0=7\\h & AD32=b8xxx\\h”);

BestMasterGenPropSet(handle, B_MGEN_REPEATMODE,
                             B_REPEATMODE_INFINITE);
printf (“generic run property infinite programmed\n”);
```

BestMasterCondStartPattSet ( ) is used to set the master trigger pattern. The B_MGEN_RUNMODE property is set to B_RUNMODE_WONDELAY to enable conditional starting.

BestMasterGenPropGet ( ) can be used to read the current generic properties.

Block Run

To start the transactions on the bus either *BestMasterBlockRun ( ) on page 171*, or *BestMasterBlockPageRun ( ) on page 172* must be called. The block run function runs the master block that is currently defined in the preparation register. The page run function runs the specified page, which may contain one or more programmed blocks.

The following function call runs block page 1:

```
BestMasterBlockPageRun(handle,0x01);
```

A program may wait until the transactions have completed as follows (see *BestStatusRegGet ( ) on page 252*):

```
/* wait until master has stopped running*/
do
{
  BestStatusRegGet(handle, &statusreg); CHECK
}
while (statusreg & 0x01);
```

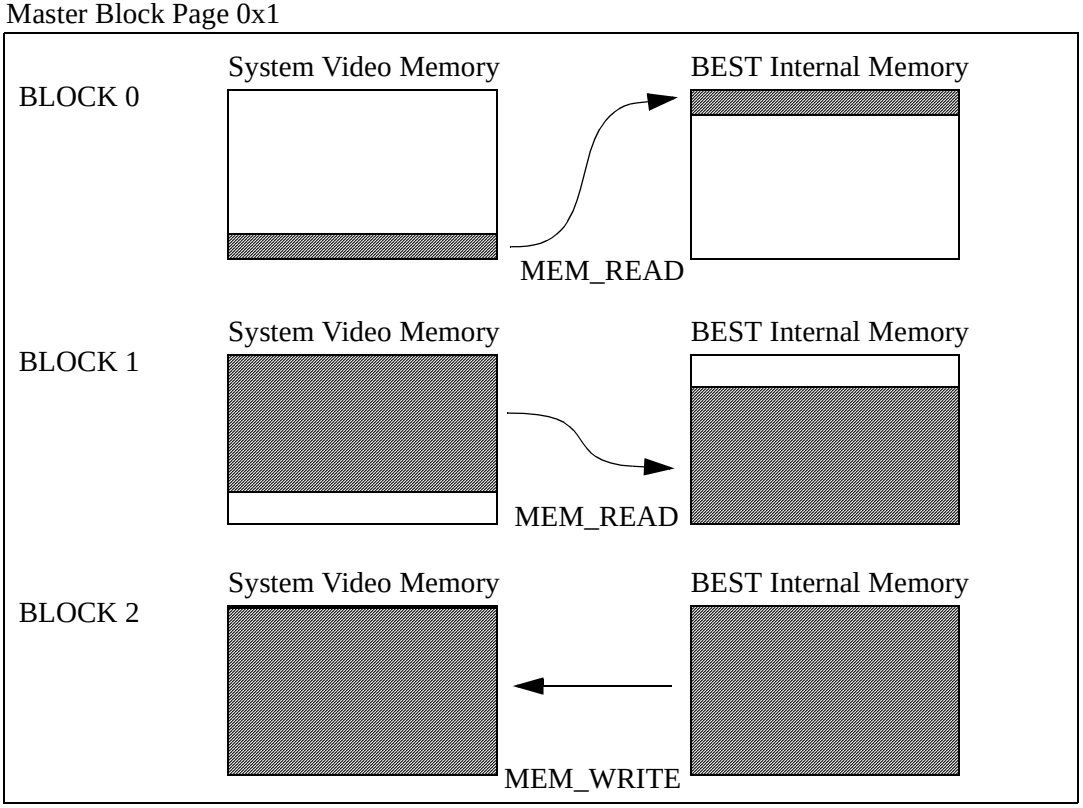
The master may be stopped using *BestMasterStop ( )*.

Complete Master Programming Example

The following example scrolls the screen, until a character is typed from the keyboard.

The example uses 5 protocol attribute phases using 1, 3, 5, 7 and 9 wait states repeatedly

It uses 3 master block transfers which are executed consecutively (chained).



Block 0, performs a memory read of the last system video memory line to internal address 0.

Block 1, reads the rest of the video memory, into internal memory below the first line.

Block 3 writes the new internal memory video image, to system video memory.

Master Transaction Example

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

/* open the communication session to card */
status=BestOpen(&handle,B_PORT_PARALLEL,B_PORT_LPT2); CHECK
status=BestConnect (handle); CHECK

/* Application program starts here */

/* master block attribute page 0x01: set protocol behavior */
status=BestMasterAttrPageInit(handle,0x01); CHECK
status=BestMasterAttrPropDefaultSet(handle); CHECK

  status=BestMasterAttrPropSet(handle,B_M_WAITS,1); CHECK
  status=BestMasterAttrPhaseProg(handle); CHECK

  status=BestMasterAttrPropSet(handle,B_M_WAITS,3); CHECK
  status=BestMasterAttrPhaseProg(handle); CHECK

  status=BestMasterAttrPropSet(handle,B_M_WAITS,5); CHECK
  status=BestMasterAttrPhaseProg(handle); CHECK

  status=BestMasterAttrPropSet(handle,B_M_WAITS,7); CHECK
  status=BestMasterAttrPhaseProg(handle); CHECK

  status=BestMasterAttrPropSet(handle,B_M_WAITS,9); CHECK
  status=BestMasterAttrPropSet(handle,B_M_DOLOOP,1); CHECK
  status=BestMasterAttrPhaseProg(handle); CHECK

/* master block page 0x01, block 0: read last line to internal adress
0x00*/
  status=BestMasterBlockPageInit(handle,0x01); CHECK
  status=BestMasterBlockPropDefaultSet(handle); CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_BUSADDR,0xb8e60);
CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_INTADDR,0x00); CHECK

status=BestMasterBlockPropSet(handle,B_BLK_BUSCMD,B_CMD_MEM_READ);
CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_NOFDWORDS,0x28); CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_ATTRPAGE,0x1); CHECK
  status=BestMasterBlockProg(handle); CHECK
  printf ("master block page 0x01 block 0 programmed\n");
```

```
/* master block page 0x01, block 1: read whole page to internal adress
0xA0 */
  status=BestMasterBlockPropDefaultSet(handle); CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_BUSADDR,0xb8000);
CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_INTADDR,0xa0); CHECK

status=BestMasterBlockPropSet(handle,B_BLK_BUSCMD,B_CMD_MEM_READ);
CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_NOFDWORDS,0x3c0);
CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_ATTRPAGE,0x1); CHECK
  status=BestMasterBlockProg(handle); CHECK
  printf ("master block page 0x01 block 1 programmed\n");

/* master block page 0x01, block 2: write whole page to adress 0xb8000 */
  status=BestMasterBlockPropDefaultSet(handle); CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_BUSADDR,0xb8000);
CHECK

status=BestMasterBlockPropSet(handle,B_BLK_BUSCMD,B_CMD_MEM_WRITE)
; CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_NOFDWORDS,0x3e8);
CHECK
  status=BestMasterBlockPropSet(handle,B_BLK_ATTRPAGE,0x1); CHECK
  status=BestMasterBlockProg(handle); CHECK
  printf ("master block page 0x1 block 0x02 programmed\n");

/* master generic run property: infinite run, conditional start */
status=BestMasterGenPropDefaultSet(handle); CHECK
status=BestMasterGenPropSet(handle, B_MGEN_RUNMODE,
  B_RUNMODE_WONDELAY); CHECK
status=BestMasterCondStartPattSet(handle, "b_state=3\\h &
    CBE3_0=7\\h & AD32=b8xxx\\h"); CHECK
status=BestMasterGenPropSet(handle,B_MGEN_REPEATMODE,
    B_REPEATMODE_INFINITE); CHECK

/* master block page run */
status=BestMasterBlockPageRun(handle,0x01); CHECK
printf ("Press ENTER\n"); getchar ();

/* Stop master running */
status=BestMasterStop(handle); CHECK
/* disconnect the port from exclusive access*/
status=BestDisconnect (handle); CHECK
/* disconnect and close the session */
status=BestClose(handle); CHECK
}
```

Creating Target Transactions

Responding as a target to master transactions involves the following:

1. Programming Target Protocol Behavior
2. Defining Target Generic Run Properties
3. Setting-up and enabling a Decoder

Programming Target Protocol Behavior

Target protocol attributes for a block transfer are defined within target attribute memory. As with master attribute memory this memory consists of 8k lines or phases which can be divided up into a maximum of 256 pages. For more info on memory organization see “Target Programming” on page 122.

Protocol attributes are programmed on a per phase basis, where each attribute memory line corresponds to a target data phase. For each data phase you can program the amount of waits, signal PERR#, signal SERR#, invert parity and program type of termination. If the DOLOOP attribute is set the current page is repeated. For more information on attributes and their default values see “b_tattrproptype” on page 204.

Attribute memory page zero cannot be overwritten, and contains all default attributes values (zero), but with the loop bit set.

After an attribute page is programmed it can be selected.

After power up, all attribute memories are programmed with zero, with the exception of DOLOOP bit which is set to one. This means any attribute page can be referenced from the selected without disastrous consequences.

To program an attribute page:

1. Call *BestTargetAttrPageInit ( )* once, with the attribute page number you want to program. This page number will be selected as the target attribute page after it has been programmed.
2. Call *BestTargetAttrPropDefaultSet ( )* once, to set the preparation register to defaults (no loop)
3. Call *BestTargetAttrPropSet ( )* once for each attribute you want to change in the preparation register

4 Call *BestTargetAttrPhaseProg ( )* once to program the attribute memory line with the content of the preparation register. This automatically increments the programming pointer to the next attribute memory line.

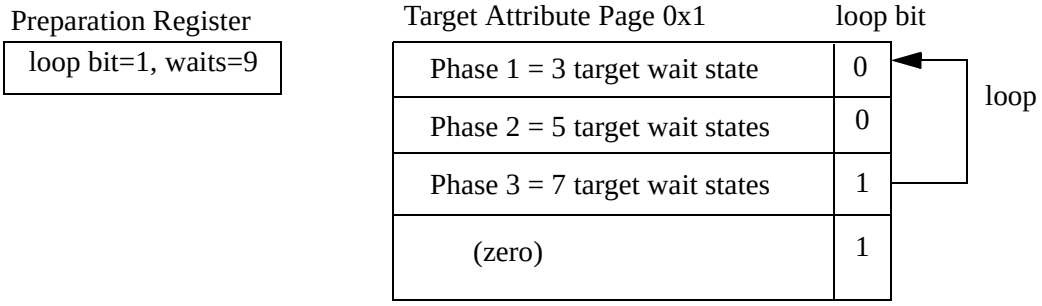
5 Repeat steps 3 and 4 for each line (data phase) you want to program. Remember to set the DOLOOP bit in the last line, in order to loop the structure.

6 Call *BestTargetAttrPageSelect ( )* to select the programmed attribute page as the target attribute page to use.

For example, the following source code extract programs 3 attribute phases, with increasing amount of wait states (no error handling shown):

```
/* Target block attribute page 0x01: set protocol behavior */
BestTargetAttrPageInit(handle,0x01);
BestTargetAttrPropDefaultSet(handle);

BestTargetAttrPropSet(handle,B_T_WAITS,3);
BestTargetAttrPhaseProg(handle);

BestTargetAttrPropSet(handle,B_T_WAITS,5);
BestTargetAttrPhaseProg(handle);

BestTargetAttrPropSet(handle,B_T_WAITS,7);
BestTargetAttrPropSet(handle,B_T_DOLOOP,1);
BestTargetAttrPhaseProg(handle);

BestTargetAttrPageSelect(handle, 0x01);
```

This leaves the following in the preparation register and in target attribute memory



Generic Run Properties

Generic run properties define how block transfers are to be executed. They define the run mode (restart from the beginning of the page with every address phase or loop attribute page only at the end of the page) and whether the memory space and I/O space decoders are enabled.

The following example sets the target attribute structure to restart from the beginning of the page with every address phase and enables the memory space decoders.

```
/* Target generic run property */
BestTargetGenPropDefaultSet(handle);
BestTargetGenPropSet(handle, B_TGEN_RUNMODE,
                             B_RUNMODE_ADDRRESTART);
BestTargetGenPropSet(handle, B_TGEN_IOSPACE, 1);
```

The "BestTargetGenPropDefaultSet ( )" function sets the target generic properties to their default values and "BestTargetGenPropSet ( )" changes these values.

Setting-up and Enabling a Target Decoder

Decoder properties define how master accesses are decoded. They determine whether the decoder should respond to memory or I/O commands, the size of the decoded address space, the PCI base address and the decoding speed.

The following example sets up decoder 2 to decode I/O address space between 0xFCE0 and 0xFCFF with a medium speed.

```
//Set up and enable decoder 2
BestTargetDecoderRead(handle, 2);
BestTargetDecoderPropSet(handle,2,B_DEC_MODE,B_MODE_IO);
BestTargetDecoderPropSet(handle,2,B_DEC_BASEADDR,0x0);
BestTargetDecoderPropSet(handle,2,B_DEC_SIZE,5);
BestTargetDecoderPropSet(handle,2,B_DEC_BASEADDR,0xFCE0);
BestTargetDecoderPropSet(handle,2,B_DEC_SPEED, B_DSP_MEDIUM);
BestTargetDecoderProg(handle, 2);

//enable decoder
BestTargetGenPropSet(handle, B_TGEN_IOSPACE, 1);
```

The base address of a decoder must be set to 0 before the decoder size can be changed.

The *BestTargetDecoderPropSet ( )* function sets the target decoder property values and these values are programmed by *BestTargetDecoderProg ( )*. The calls to *BestTargetDecoderPropSet ( )* may be replaced with a single call to *BestTargetDecoder1xProg ( )* shown in the following example.

```
//Set up and enable decoder 1
BestTargetDecoder1xProg(handle,2,B_MODE_IO,5,0xFCE0,

B_DSP_MEDIUM,B_LOC_SPACE32,0);


//enable decoder
BestTargetGenPropSet(handle, B_TGEN_IOSPACE, 2);
```

The *BestAllPropStore ( )* function can be used to save the decoder properties (and all other properties) as powerup defaults.

Programming a termination

The target can be programmed to terminate a transaction by setting the *B_T_TERM (term)* target protocol attribute to a value other than B_TERM_NOTERM. The following example signals a target abort on the third data phase:

```
//Program Target Protocol Behavior
/* Target block attribute page 0x01: set protocol behavior */
BestTargetAttrPageInit(handle,0x01);
BestTargetAttrPropDefaultSet(handle);

  BestTargetAttrPropSet(handle,B_T_WAITS,3);
  BestTargetAttrPhaseProg(handle);

  BestTargetAttrPropSet(handle,B_T_WAITS,5);
  BestTargetAttrPhaseProg(handle);

  BestTargetAttrPropSet(handle,B_T_WAITS,7);
  BestTargetAttrPropSet(handle,B_T_TERM,B_TERM_DISCONNECT);
  BestTargetAttrPhaseProg(handle);

  BestTargetAttrPropSet(handle,B_T_WAITS,5);
  BestTargetAttrPropSet(handle,B_T_TERM,B_TERM_ABORT);
  BestTargetAttrPhaseProg(handle); check

  BestTargetAttrPropSet(handle,B_T_DOLOOP,1);
  BestTargetAttrPhaseProg(handle);
BestTargetAttrPageSelect(handle, 0x01);
```

Target Transaction Example

This example makes I/O address space 0xFCE0 - 0xFCFF available for reading and writing. Data may be writen to or read from this address space by writing a simple program on the DUT. For example, if the DUT is running DOS, the following code fragment may be typed in the debug utility:

```
- a
1D10:0100 mov dx, fce8
1D10:0103 mov ax, ff3c
1D10:0106 out dx, ax
1D10:0107
-
```

Alternatively the host to PCI access functions may be used to download code to the DUT (see “Using the Host access functions” on page 67.).

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

#define CHECK  if (status != B_E_OK) {printf (“%s\n”,
BestErrorStringGet(status));return -1;}

int main ( )
{

b_errtype status;
b_handletype handle;


/* open the communication session to card */
status=BestOpen(&handle,B_PORT_RS232,B_PORT_COM1); CHECK

status=BestConnect (handle); CHECK

/* Application program starts here */

//Disable IO decoders while programming
status=BestTargetGenPropSet(handle, B_TGEN_IOSPACE, 0); CHECK

//Program Target Protocol Behavior
/* Target block attribute page 0x01: set protocol behavior */
status=BestTargetAttrPageInit(handle,0x01); CHECK
status=BestTargetAttrPropDefaultSet(handle); CHECK
```

```
  status=BestTargetAttrPropSet(handle,B_T_WAITS,3); CHECK
  status=BestTargetAttrPhaseProg(handle); CHECK

  status=BestTargetAttrPropSet(handle,B_T_WAITS,5); CHECK
  status=BestTargetAttrPhaseProg(handle); CHECK

  status=BestTargetAttrPropSet(handle,B_T_WAITS,7); CHECK
  status=BestTargetAttrPhaseProg(handle); CHECK

  status=BestTargetAttrPropSet(handle,B_T_WAITS,9); CHECK
  status=BestTargetAttrPropSet(handle,B_T_DOLOOP,1); CHECK
  status=BestTargetAttrPhaseProg(handle); CHECK
status=BestTargetAttrPageSelect(handle, 1); CHECK

/* Target generic run property */
status=BestTargetGenPropDefaultSet(handle); CHECK
status=BestTargetGenPropSet(handle, B_TGEN_RUNMODE,
B_RUNMODE_ADDRRESTART); CHECK

//Set up and enable decoder 2 */
/*Set properties in decoder preparation register */
status=BestTargetDecoderPropSet(handle,B_DEC_MODE,B_MODE_IO);
CHECK
status=BestTargetDecoderPropSet(handle,B_DEC_BASEADDR,0); CHECK
status=BestTargetDecoderPropSet(handle,B_DEC_SIZE,5); CHECK
status=BestTargetDecoderPropSet(handle,B_DEC_BASEADDR,0xFCE0);
CHECK
status=BestTargetDecoderPropSet(handle,B_DEC_SPEED, B_DSP_MEDIUM);
CHECK
//Program decoder using properties in decoder preparation register
status=BestTargetDecoderProg(handle, 2); CHECK
//enable IO decoders
status=BestTargetGenPropSet(handle, B_TGEN_IOSPACE, 1); CHECK


/* disconnect and close the session */
status=BestDisconnect(handle); CHECK
status=BestClose(handle); CHECK
}
```

Triggering the Analyzer Trace Memory

Triggering the trace memory involves:

1. Setting the Sample Qualifier (optional)
2. Setting the trigger pattern
3. Running the analyzer
4. Uploading the captured data
5. Interpreting the captured data

Setting the Sample Qualifier (optional)

This means defining a pattern which is used to qualify each bus state to determine if the state is stored or not stored. If the sample qualifier pattern is true, then the state is stored in trace memory. This is done using the *BestTracePattPropSet ( )* function with the B_PT_SQ property option. For example, the following sample qualifier stores all states:

```
BestTracePattPropSet (handle, B_PT_SQ, “1”);
```

Setting the Trigger Pattern

We now define the bus pattern which triggers the storing of data in the analyzer trace memory.

For example, the following extract triggers on a memory write address phase:

```
BestTracePattPropSet (handle, B_PT_TRIGGER,
   “b_state=3\\h & CBE3_0==7\\h & AD32=b8xxx\\h”);
```

To trigger on a protocol violation, the following can be used:

```
BestTracePattPropSet (handle, B_PT_TRIGGER,
   “b_err=1”);
```

Running the analyzer

BestAnalyzerRun ( ) enables the analyzer to start acquiring data:

```
BestAnalyzerRun (handle);
```

BestAnalyzerStop ( ) can be used to stop the analyzer. These two functions affect the trace data and protocol observer. *BestTraceRun ( )* and *BestTraceStop ( )* can be used to enable and disable the trace memory. After calling BestAnalyzerStop or BestTraceStop, *BestTraceStatusGet ( )* can be used to get the current status of the trace memory. For example:

```
b_int32 tracestatus;
BestTraceStatusGet (handle, B_TRC_LINESCAPT, &tracestatus);
```

returns the number of lines captured. After calling BestAnalyzerStop or BestTraceStop, BestTraceStatusGet can be used to determine the trigger point and number of captured lines.

```
b_int32 trigpoint, numlines;
BestTraceStatusGet (handle, B_TRC_LINESCAPT, &trigpoint);
BestTraceStatusGet (handle, B_TRC_TRIGPOINT, &numlines);
```

Uploading captured data

BestTraceDataGet ( ) uploads data captured by the the analyzer to the host:

```
err = BestTraceDataGet(handle, disp_start, lines, data); CHECK
```

Interpreting captured data

Since future releases may provide captured data in a different format to the current format, *BestTraceBitPosGet ( )* and *BestTraceBytePerLineGet ( )* are provided to extract information from the captured data, for example:

```
err = BestTraceBytePerLineGet(handle, &bytes_per_line); CHECK
err = BestTraceBitPosGet(handle, B_SIG_AD32, &ad32_pos, &ad32_len);
      CHECK
```

The above code fragment determines the number of bytes in each line (bus cycle) and determines the position and size of the AD_32 bus within the captured data. The following fragment determines if IRDY was asserted in a specific bus cycle:

```
err = BestTraceBitPosGet(handle, B_SIG_IRDY, &irdy_pos, &irdy_len);
if(data[i + irdy_pos/32]>>(irdy_pos%32)) & 1)
{
  printf("IRDY was asserted in cycle %d", i);
}
```

Programming the Analyzer

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

```
#define CHECK  if (status != B_E_OK) {printf (“%s\n”,
BestErrorStringGet(status));return -1;}

int main ( )
{

  b_errtype status;
  b_handletype handle;
char buffer[256];

  /* open the communication session to card */
  status=BestOpen(&handle,B_PORT_RS232,B_PORT_COM1); CHECK

  status=BestConnect (handle); CHECK

  /* Application program starts here */
//
// Set up Pattern
//
  printf(“Enter trigger pattern (e.g. !FRAME: “);
  gets(buffer);
  if (*buffer)
  {
    err = BestTracePattPropSet(handle, B_PT_TRIGGER, buffer); CHECK
  }

//
// Run the Analyzer
//

  err=BestAnalyzerRun(handle); CHECK

//
// Run some tests using the Master Transfers/Built-in test functions
//

// .........

//
// Upload trace data
//

  err = BestTraceStatusGet(handle, B_TRC_STAT, &stat); CHECK
  while ((stat & 3) != 3)
  {
    // Analyzer hasn’t triggered yet
    // or it hasn’t stooped acquiring data.
```

```
  }
  err = BestAnalyzerStop(handle); CHECK
  err = BestTraceStatusGet(handle, B_TRC_TRIG_POINT, &trig); CHECK
  disp_start = trig -3;
  if (disp_start > trig) // check for unsigned int underflow
    disp_start =0;
  err = BestTraceDataGet(handle, disp_start, lines, data); CHECK

/*
Interpret the Trace Data
                              */

  err = BestTraceBytePerLineGet(handle, &bytes_per_line); CHECK
  err = BestTraceBitPosGet(handle, B_SIG_AD32, &ad32_pos, &ad32_len);
        CHECK
  err = BestTraceBitPosGet(handle, B_SIG_CBE3_0, &cbe_pos, &cbe_len);
        CHECK
  err = BestTraceBitPosGet(handle, B_SIG_FRAME, &frame_pos,
        &frame_len); CHECK
  err = BestTraceBitPosGet(handle, B_SIG_IRDY, &irdy_pos, &irdy_len);
        CHECK
  err = BestTraceBitPosGet(handle, B_SIG_TRDY, &trdy_pos, &trdy_len);
        CHECK
  err = BestTraceBitPosGet(handle, B_SIG_DEVSEL, &devsel_pos,
        &devsel_len); CHECK
  err = BestTraceBitPosGet(handle, B_SIG_STOP, &stop_pos, &stop_len);
        CHECK

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

/* disconnect and close the session */
status=BestDisconnect(handle); CHECK
status=BestClose(handle); CHECK
}
```

Programming the Protocol Observer

Programming the protocol observer involves:

1. Setting the Observer Properties
2. Setting the Observer Mask
3. Running the Observer
4. Getting the Protocol Errors

Setting the Observer Properties

BestObsPropDefaultSet ( ) sets the Observer Properties to their default values:

```
err = BestObsPropDefaultSet(handle); CHECK
```

Setting the Observer Mask

Define the protocol rules to ignore, for example:

```
err = BestObsMaskSet(handle, B_R_PARITY_1 | B_R_PARITY_2, 1);
```

This function is used to mask out individual protocol errors and is described in *BestObsMaskSet ( ) on page 210*. *BestObsMaskGet ( )* can be used to retrieve the current observer mask. For a definition of each error, see “Protocol Observer” on page 141.

Running the Observer

The Observer may be run and stopped using “BestObsRun ( )” and “BestObsStop ( )” respectively:

```
err = BestObsRun(handle);
// ... Run master transactions etc or run analyzer and wait for trigger
err = BestObsStop(handle);
```

Getting the Protocol Errors

The *BestObsStatusGet ( )* function can be used to determine if there were any errors:

```
err = BestObsStatusGet(handle, B_OBS_FIRSTERR, &value); CHECK
```

BestObsStatusClear ( ) clears the observer status register. When a protocol error occurs, and it is not masked, the Observer Status Register (bit 2) is set, and the appropriate bit in the Accumulated error register is set (see section Accumulated Error Register and First Error Register (accuerr and firsterr) on page 215). The following extract determines if error ‘i’ has occured and prints the error:

```
err = BestObsStatusGet(handle, B_OBS_FIRSTERR, &value); CHECK
err = BestObsStatusClear(handle);
// ...
  if (value & (1<<i))
  {
    err = BestObsErrStringGet(handle, i, &errstring); CHECK
    printf(“Protocol error: %s\n”, errstring);
  }
```

To translate the returned value into a meaningful text string, use function *BestObsErrStringGet ( )*.

NOTE: The first protocol violation which occurs will be displayed on the hex display.

Example

This example sets up and runs the protocol observer and prints a description of each protocol violation flagged by the protocol observer.

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


#define CHECK  if (status != B_E_OK) {printf (“%s\n”,
BestErrorStringGet(status));return -1;}

int main ( )
{

  b_errtype status;
  b_handletype handle;


  /* open the communication session to card */
  status=BestOpen(&handle,B_PORT_RS232,B_PORT_COM1); CHECK

  status=BestConnect (handle); CHECK
```

```
  /* Application program starts here */
  #define NO_OF_PROTOCOL_RULES 25

  err = BestObsPropDefaultSet(handle); CHECK
  err = BestObsMaskSet(handle, B_R_PARITY_2, 1); CHECK
  err = bestObsStatusClear(handle); CHECK
  err = BestObsRun(handle); CHECK


// ...

// Run some transactions

// ...

  err = BestObsStop(handle); CHECK
  err = BestObsStatusGet(handle, B_OBS_OBSSTAT, &value); CHECK
  if ((value & 4) == 0)
  {
    printf("No protocol error occured\n");
    return B_E_OK;
  }
  err = BestObsStatusGet(handle, B_OBS_FIRSTERR, &value); CHECK
  for (i = 0; i < NO_OF_PROTOCOL_RULES; i++)
  {
    if (value & (1<<i))
    {
      err = BestObsErrStringGet(handle, i, &errstring); CHECK
      printf("Protocol error: %s\n", errstring);
    }
  }

/* disconnect and close the session */
status=BestDisconnect(handle); CHECK
status=BestClose(handle); CHECK
}
```

Using the Host access functions

Communicating with the DUT using the host to PCI access functions involves:

1 Setting the Master Generic Properties

2 Preparing for the transfer

3 Performing the transfer

See *BestHostPCIRegGet ( ) on page 288*, *BestHostPCIRegSet ( ) on page 286* for information on how to perform single I/O, memory or configuration space accesses. *BestHostIntMemFill ( )* and *BestHostIntMemDump ( )* can be used to access the internal memory of the HP E2925A card.

Setting the Master Generic Properties

The master generic properties are used when data is transferred between the host and the DUT (see “BestMasterGenPropSet ( )” on page 185.). The following line ensures that the master state-machine transfers data to the DUT just once.

```
err = BestMasterGenPropSet(handle, B_MGEN_REPEATMODE,
                           B_REPEATMODE_SINGLE); CHECK
```

Preparing for the transfer

Before performing a memory transfer the *BestHostSysMemAccessPrepare ( )* function should be called to set up the block transfers used to transfer data to the DUT.

This example sets a block size of 64 and speciifes a memory read bus command:

```
err = BestHostSysMemAccessPrepare(handle, B_CMD_MEM_READ, 64);
```

Performing the transfer

Start transferring data between the host and system memory. “BestHostSysMemFill ( )” and “BestHostSysMemDump ( )” may be used to transfer data between the host and the DUT. These functions may be used to transfer test code to the DUT, for example. See also *Host to PCI Access Functions on page 297*.

```
BestHostSysMemFill(handle, bus_addr, NUM_BYTES, BLK_SIZE, buffer2);
```

Example

This example transfers random data between the host and the video memory of the DUT:

```
#include <stdio.h>
#include <mini_api.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;
  int i
  b_int8 buffer1[NUM_BYTES];
  b_int8 buffer2[NUM_BYTES];
  b_int32 bus_addr = 0xb8000;

  /* open the communication session to card */
  status=BestOpen(&handle,B_PORT_RS232,B_PORT_COM1); CHECK
  status=BestConnect (handle); CHECK

  /* Application program starts here */
  /* fill a buffer with random data */
  for (i = 0; i < NUM_BYTES; i++)
    buffer2[i] = rand();

  err = BestMasterGenPropSet(handle, B_MGEN_REPEATMODE,
                             B_REPEATMODE_SINGLE); CHECK

  /* prepare for BestHostSysMemDump */
  err = BestHostSysMemAccessPrepare(handle, B_CMD_MEM_READ,
         BLK_SIZE); CHECK
  err = BestHostSysMemDump(handle, bus_addr, NUM_BYTES, BLK_SIZE,
                             buffer1); CHECK

  /* prepare for BestHostSysMemFill */
  err = BestHost SysMemAccessPrepare(handle,
           B_CMD_MEM_WRITE,BLK_SIZE); CHECK
  err = BestHostSysMemFill(handle, bus_addr, NUM_BYTES, BLK_SIZE,
                             buffer2); CHECK

/* disconnect and close the session */
status=BestDisconnect(handle); CHECK
status=BestClose(handle); CHECK
}
```

Using the CPU port

The CPU port provides a simple parallel programming interface. It provides 16 address lines, 16 data lines, control, two chip selects, two byte enables and an interrupt line. The byte enable lines allow 8 or 16-bit accesses. See *CPU Port on page 310* for detailed information on the CPU port.

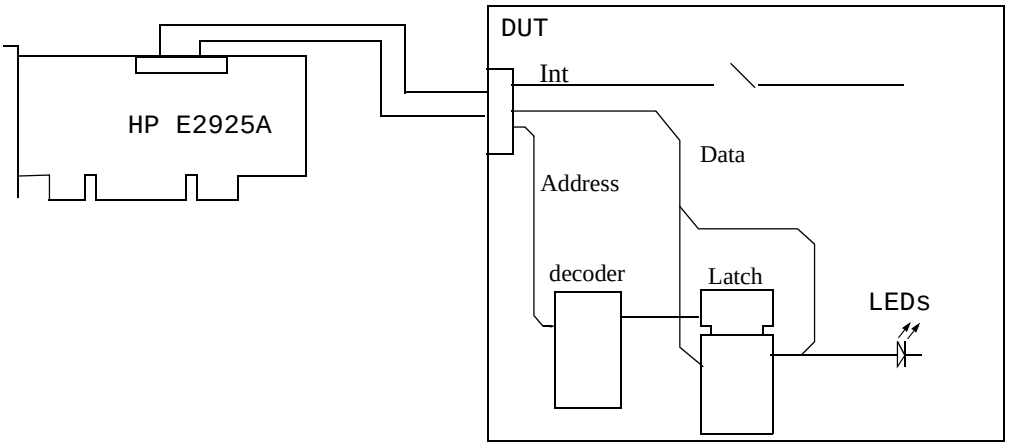
Using the CPU port involves:

- providing an interface to the DUT
- enabling the CPU port
- waiting for an interrupt to occur (optional)
- reading from and writing to the port
- disabling the CPU port

Providing an interface to the DUT

In this example we use the CPU port to drive 8 LEDs. We generate interrupts using a switch and debouncing circuit. When an interrupt occurs, the program reads an 8-bitvalue from the device, increments it and writes the new value on the 8 low end data lines.

Figure 2 Circuit used in this example



Enabling the CPU port

In this example we set the CPU port mode to master, set protocol type to Intel compatible and set ready generation to internal. Setting CPU port mode to master enables the CPU port. When we have finished using the CPU port we will disable it by setting the B_CPU_MODE property to B_CM_DISABLED. RDY# signal generation is set to internal. This means that the E2925 card will not wait for the RDY# signal to be asserted. Instead, it assumes that the device is ready after 300ns. The protocol type is set to Intel compatible (the only protocol currently available).

```
err = BestCPUportPropSet(handle, B_CPU_MODE, B_CM_MASTER); CHECK
```

Waiting for an interrrupt to occur

The *BestCPUportIntrStatusGet ( )* function is used to read the interrrupt latch. In the following example we wait until the interrupt has triggered and clear the latch when it has triggered.

```
while(!intr)
{
  err = BestCPUIntrStatusGet(handle, &intr); CHECK
}
BestCPUIntrClear(handle); CHECK
```

In this example a simple switch and debounce circuit is used to drive the interrupt line.

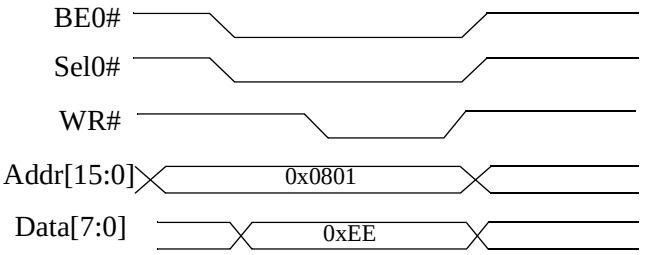
Write data

The *BestCPUportWrite ( )* function is used to write data to the CPU port. Bytes or words can be written to the CPU port. The size of the data is specified by the **size** parameter.

```
BestCPUportWrite(handle, 0, 0x0801, 0xEE, B_SIZE_BYTE); CHECK
```

NOTE: If using word accesses the address must be word-aligned.

This asserts sel0#, WR#, BE0# and drives A[15:0] and D[7:0] as shown in the following diagram



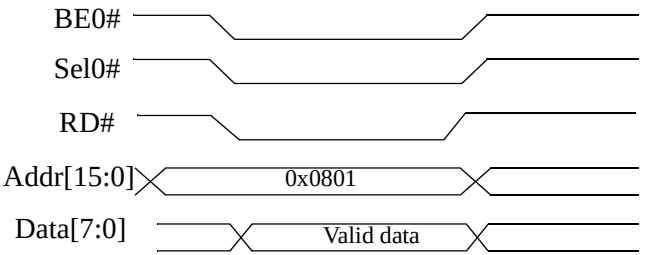
Read data

The *BestCPUportRead ( )* function is used to read data from the CPU port. Bytes or words can be written to the CPU port. The size of the data is specified by the **size** parameter.

```
b_int32 data;
BestCPUportRead(handle, 0, 0x0801, &data, B_SIZE_BYTE); CHECK
```

NOTE: If using word accesses the address must be word-aligned.

This asserts sel0#, RD#, BE0# and drives A[15:0] and D[7:0] as shown in the following diagram



Example

This example writes a byte to the CPU port. It then polls the interrupt latch. When an interrupt occurs it clears the latch and reads a byte from the CPU port.

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

#define CHECK  if (status != B_E_OK) {printf (“%s\n”, BestErrorString
Get(status));return -1;}

int main ( )
{
  b_int32 i = 0;
  b_int32 intr = 0;
  b_errtype err;
  b_handletype handle;
  /* open the communication session to card */
  status=BestOpen(&handle,B_PORT_RS232,B_PORT_COM1); CHECK
  status=BestConnect (handle); CHECK
  /* Enable CPU port*/
  err = BestCPUportPropSet(handle, B_CPU_MODE, B_CM_MASTER); CHECK
  err = BestCPUportPropSet(handle, B_CPU_PROTOCOL, B_CP_INTEL); CHECK
  err = BestCPUportPropSet(handle, B_CPU_RDYTYPE, B_CR_AUTO); CHECK
  err = BestCPUportRST(handle, 0); CHECK
  err = BestCPUportRST(handle, 1); CHECK
  while(1)
  {
    BestCPUportWrite(handle, 0, 0x0801, i, B_SIZE_BYTE); CHECK
    while(!intr)
    {
      err = BestCPUIntrStatusGet(handle, &intr); CHECK
    }
    BestCPUIntrClear(handle); CHECK
    BestCPUportRead(handle, 0, 0x0801, &i, B_SIZE_BYTE); CHECK
    i++;
    i %= 256;
  }
  err = BestCPUportPropSet(handle, B_CPU_MODE, B_CM_DISABLED); CHECK
  err = BestDisconnect(handle);
  err = BestClose(handle);
}
```

Using the Static I/O port

The static I/O port provides 8 independent I/O pins. Each pin can be indiviually programmed as input, open-drain output or totem pole output.

1 Set pin properties (e.g. specify input/output pins)

2 Read/Write port or pin.

Setting pin properties

By default all pins are input pins. A pin may be configured as an output pin using *BestStaticPropSet ( ) on page 234*. BestStaticPropSet configures each pin of the 8 Bit static IO port as either:

- Input only
- Open-drain
- Totem-pole output

The following extract configures pin 2 as a totem-pole output:

```
status = BestStaticPropSet(handle, 2, B_STAT_PINMODE,
            B_PMD_TOTEMPOLE); CHECK
```

Reading from and writing to a port or pin

Individual pins can be written using *BestStaticPinWrite ( ) on page 238*. The second parameter is the pin to write to and the third is the value to be written to the port (0 or 1).

The following extract writes a value of 1 to pin 2:

```
status = BestStaticPinWrite(handle, 2, 1); CHECK
```

The entire static I/O port can be read from or written to using *BestStaticRead ( ) on page 237* and *BestStaticWrite ( ) on page 236*, respectively.

```
b_int32 value;

for(i = 0; i < 4; i++)
    status = BestStaticPropSet(handle, 2, B_STAT_PINMODE,
                B_PMD_TOTEMPOLE); CHECK
for(i = 4; i < 8; i++)
    status = BestStaticPropSet(handle, 2, B_STAT_PINMODE,
```

```
                B_PMD_INPONLY); CHECK
BestStaticWrite(handle,0x0F)
BestStaticRead(handle, &value)
printf("Value input at static IO port (pins 7:4) %i", value>>4);
```

Example

This example sets and clears each pin in turn. For more information on Static IO, see "Static I/O Port" on page 315.

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

/*
   This C-API example "Using the Static I/O port"
    in the HP E2925A User's Guide
 */

#define CHECK  {\
  if(err != B_E_OK) {\
    printf("%s\n", BestErrorStringGet(status));\
    if(err == B_E_FUNC) \
      BestSMReset(handle);\
    return -1; \
  }           \
}

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

  b_errtype err;
  b_handletype handle;
  b_int32 devid;
  int i, j;

  /*Initialize port internal structs and variables*/
  /*  Get device number */
  status = BestDevIdentifierGet(0x103c, 0x2925, 0, &devid); CHECK;
  /* get handle */
  err = BestOpen(&handle, B_PORT_PCI_CONF, devid); CHECK;

  /* Establish Connection to card */
  err = BestConnect ( handle ); CHECK;

  /* Application program goes in here, for example:*/
  /* Configure all pins as totem pole outputs:*/
```

```
  for(i = 0; i < 8; i++)
  err=BestStaticPropSet(handle,i,B_STAT_PINMODE,B_PMD_TOTEMPOLE);
CHECK

  /* Write to each pin, setting it to 1 and then 0. */
  for(j = 0; j < 256; j++)
    for(i = 0; i < 8; i++)
    {
      status = BestStaticPinWrite(handle, i, 1); CHECK
      status = BestStaticPinWrite(handle, i, 0); CHECK
    }
  /* Configure all pins as inputs:*/
  for(i = 0; i < 8; i++)
    status = BestStaticPropSet(handle, i, B_STAT_PINMODE,
B_PMD_INPONLY); CHECK

  /* disconnect from the current port*/
  status = BestDisconnect (handle); CHECK;
  /* close the session and deallocate memory*/
  status = BestClose(handle); CHECK;
  return 0;
}
```

Using the Hex display

Using the hex display involves:

1. Setting the hex display to user mode using BestDisplayPropSet
2. Writing values to the hex display using BestDisplayWrite
3. Returning the hex display to protocol observer mode

Setting the hex display to user mode

BestDisplayPropSet ( ) is used to set the mode of the hex display.

```
status = BestDisplayPropSet(handle, B_DISP_USER); CHECK
```

The above line sets the hex display to user mode. The hex display may be reset to protocol observer mode as follows:

```
status = BestDisplayPropSet(handle, B_DISP_CARD); CHECK
```

Writing values to the hex display

BestDisplayWrite ( ) is used to write to the hex display

```
status = BestDisplayWrite(handle, 0x00ff); CHECK
```

The above line writes the value FF to the hex display.

Example

This example cycles through all values between 0 and 0xFF several times:

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

/*
  This C-API example accompanies “Using the Hex display”
   in the HP E2925A User’s Guide
 */

#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 cards*/
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 card*/
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;
}
```

Communicating with the DUT via the mailbox registers

The card provides two mailbox registers, which are part of the configuration space, for communication between programs running on the external host and the program executed by the DUT CPU.

Communicating with the DUT using the mailbox functions involves:

- Using the external interface mailbox functions on an external host.
- Using the PCI mailbox functions on the device under test.

Introduction

Both mailbox registers are unidirectional and each has an associated status bit. A variable indicates whether or not the corresponding register contains valid data. Using the mailbox access functions involves reading from/writing to the mailbox register and checking the returned status.

Using the external interface mailbox functions

The *BestMailboxSendRegWrite ( )* and *BestMailboxReceiveRegRead ( )* functions are used on the host. The following example attempts to read from the receive register until the status parameter indicates that the returned data is valid:

```
do {
  err = BestMailboxReceiveRegRead(handle, &data, &status); CHECK
} while(status == 0);
```

If the status bit is set the data value returned is a previously unread message.

Using the PCI mailbox functions

The PCI mailbox functions (*BestPCICfgMailboxSendRegWrite ( )* and *BestPCICfgMailboxReceiveRegRead ( )*) do not require a handle. The device identifier of the card is used to identify it.

The following example attempts to write to the send register until the the status bit indicates that data has been successfully written:

```
BestDevIdentifierGet(0x103C, 0x2925, 0, &devid);
do {
  err = BestPCICfgMailboxSendRegWrite(devid, &data, &status); CHECK
```

```
   } while(status == 0);
```

If the status bit is set, then the send register still contains data that has not yet been read by the other participant. If this is the case then the data is not written to the send register.

Example

The following code is executed on the external host. It reads a sequence of strings from mailbox register and prints them.

```
#include <stdio.h>
#include <stdlib.h>
#include <mini_api.h>
#include <time.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <io.h>
#include <stdio.h>

#define CHECK  if (err != B_E_OK) {printf ("%s\n", \
 BestErrorStringGet(err)); return -1; }

/*
  This C-API example accompanies "Communicating with the DUT via the
mailbox registers"
   in the HP E2925A User's Guide.
 */

int best_mboxreads(b_handletype handle, char *buffer, int buffsize)
{
  b_int32 status = 0;
  b_errtype err;
  b_int32 data;
  int i;

  for(i=0; i < buffsize; i++){
    do {
```

```
    /* Attempt to read a byte from the mailbox register */
    err = BestMailboxReceiveRegRead(handle, &data, &status); CHECK
   } while (status == 0); /* until previously unread data is
available */
   buffer[i] = data;
   if(buffer[i] == 0)
    break;
  }
  buffer[buffsize-1]=0;
  return 0;
}

int main (int argc, char *argv[] )
{
  int quit=0, i, j;
  b_errtype err;
  b_handletype handle;
  b_int32 byte, status=0, bus_addr;
  unsigned char *data, *line;
  char buffer[256];

  /* open the communication session to card */
  err=BestOpen(&handle,B_PORT_PARALLEL,B_PORT_LPT1); CHECK
  err=BestConnect (handle); CHECK

  while(1)
  {
   best_mboxreads(handle, buffer, 256);
   printf(buffer);
  }
  err=BestDisconnect(handle); CHECK
  err=BestClose(handle); CHECK
  return 0;
}
```

The following code is executed on the device under test. It sends a sequence of strings to the external host:

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

#define CHECK if (err != B_E_OK) {printf ("%s\n", \
 BestErrorStringGet(err));return -1;}


/*
  This s a C-API example accompanying "Communicating with the DUT
via the mailbox registers"
   in the HP E2925A User's Guide.
*/

/* Write a string to an external host using the mailbox register */

int mboxwrites(b_int32 devid, const char *str)
{
 b_errtype err;
 b_int32 status;
 do {
  do {
   /* Attempt to send a byte */
   err = BestPCICfgMailboxSendRegWrite(devid, *str, &status); CHECK
  } while(status == 0); /* repeat until successful */
 } while(*str++);
 return 0;
}

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

b_errtype err;
b_handletype handle;
b_int32 statusreg, devid;
int i;
char buffer[256];
```

```
/* Get Device Identifier of1st HP E2925 (index = 0) */
err=BestDevIdentifierGet(0x103C, 0x2925, 0, &devid); CHECK


/* Application program starts here */

for(i=0; i < 1000000; i++) {
  sprintf(buffer, "iteration %i\n", i);
  mboxwrites(devid, buffer);
  printf(buffer);
}


printf ("Press ENTER\n"); getchar ();

  return 0;
}
```

Programming the configuration space of the E2925A

The HP E2925A provides a programmable configuration space. API functions are provided to modify the card's configuration space and its read/write mask, to enable/disable the card's configuration space and to store these values as the powerup defaults. See also *Configuration Space on page 128*.

1. Write to configuration space registers
2. Set configuration space masks
3. Set 'confrestore' powerup property
4. Save properties as powerup defaults

Writing to the card's configuration space registers

The E2925A allows the values of its configuration space to be set using the C-API

BestConfRegSet ( ) and *BestConfRegGet ( )* are used to write and read the configuration space of the card.

The following example sets the device and vendor id of the card:

```
err=BestConfRegSet(handle, 0x00, 0x2925103c); CHECK
```

Setting configuration space masks

BestConfRegMaskSet ( ) is used to set the read/write mask of the specified register.

```
err=BestConfRegMaskSet(handle, 0x00, 0x00000000); CHECK
```

The above example makes the device and vendor id register read-only.

See also *BestConfRegMaskGet ( ) on page 249*.

Set 'confrestore' powerup property to 0

The confrestore powerup property defines how the configuration space registers are assigned values at powerup. If this property is 1 the read/writeable bits are restored from the stored powerup properties. If this property is 0 the factory default values are used for the read/write-able bits.

```
err = BestPowerupPropSet(handle, B_PU_CONFRESTORE, 0); CHECK
```

See *BestPowerUpPropSet ( ) on page 262*.

Save properties as powerup defaults

BestAllPropStore ( ) is used to store the current values of all properties as the powerup defaults.

```
err = BestAllPropStore(handle); CHECK
```

Example

The following example sets the device and vendor id of the card:

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

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

/*
 C-API example accompanying “Programming the configuration space of
the E2925A”
  in the User’s Guide
 */

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

b_errtype status;
b_handletype handle;

BestOpen(&handle,B_PORT_PARALLEL,B_PORT_LPT2);
BestConnect ( handle );

/* Set vendor and device id:*/
err=BestConfRegSet(handle, 0x00, 0x2925103c); CHECK
/* Make Device and Vensor ID read-only*/
err=BestConfRegMaskSet(handle, 0x00, 0x00000000); CHECK
/* Read/write bits will have their factory default values at powerup */
err = BestPowerupPropSet(handle, B_PU_CONFRESTORE, 0); CHECK
err = BestAllPropStore(handle); CHECK

/* disconnect from the current port*/
BestDisconnect (handle);

/* close the session and deallocate memory*/
```

```
BestClose(handle);
}
```

Using the onboard expansion ROM

The HP E2925A provides a 64k EEPROM which can be used as an expansion ROM. Using the onboard expansion ROM involves:

1. Writing data to the expansion ROM
2. Setting decoder and board properties
3. Saving properties as powerup defaults

Writing data to the expansion ROM

BestExpRomByteWrite ( ) is used to write a single byte to the expansion ROM. The second parameter is the offset of the byte within the expansion ROM. The third parameter is the byte to be written.

The following example writes 1k of data beginning at offset 0 within the onboard expansion-ROM:

```
for(i=0; i< 0x400; i++)
   err=BestExpRomByteWrite(handle, i, mem[i]); CHECK
```

See also *BestExpRomByteRead ( ) on page 251*.

NOTE: The upper 1kB of the EEPROM is used to store powerup properties.

Setting decoder and board properties

The B_BOARD_ROMUSAGE board property defines how the onboard ROM is used. A value of B_ROMUSAGE_EXTERNAL specifies that the onboard ROM should be used as an expansion ROM. It may be set using *BestBoardPropSet ( )*:

```
err=BestBoardPropSet(handle, B_BOARD_ROMUSAGE,
           B_ROMUSAGE_EXTERNAL); CHECK
```

The above line enables the onboard EEPROM for use as an expansion ROM. If B_BOARD_ROMUSAGE has a value of B_ROMUSAGE_INTERNAL the EEPROM will be used to store master and target attributes.

The following code sets up the expansion ROM decoder to decode a 256kB address space:

```
err = BestTargetDecoderPropSet(handle, B_DEC_BASEADDR, 0); CHECK
err = BestTargetDecoderPropSet(handle, B_DEC_SIZE, 18); CHECK
```

B_DEC_SIZE can have a value of 0 or 18. Addresses > 64k are mapped to an EEPROM address between 0 and 64k.

Saving properties as powerup defaults

BestAllPropStore ( ) is used to store the current values of all properties as the powerup defaults. The next time the device under test is powered up, the expansion ROM will be visible to the BIOS.

```
err = BestAllPropStore(handle); CHECK
```

Example

The following code sets up the expansion ROM header and reads the POST and runtime code and data from several files.

```
#include <stdio.h>
#include <stdlib.h>
#include <mini_api.h>
#include <time.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>


#define CHECK  if (err != B_E_OK) {printf (“%s\n”, \
 BestErrorStringGet(err)); return -1; }

/* This C-API example accompanies “Using the onboard expansion ROM”
    in the HP E2925A User’s Guide.
*/

#define EXPROMCODEOFFSET 0x80 /* Start of code within ROM image */

/* #define some ROM header offsets */

#define ROMSIG1OFFSET 0
#define ROMSIG2OFFSET 1
#define INITSIZEOFFSET 2
#define INITPOINTEROFFSET 3
#define PCISTRUCTPOINTEROFFSET 0x18

#define INITPOINTER 0x000300   /* location of initialization code */
```

```
#define PCISTRUCTPOINTER 0x20  /* location of PCI structure */
#define ROMHEADERSIZE 0x20

/* PCI structure - offsets*/
#define PCISTRUCTSIGOFFSET 0
#define PCIVENDORIDOFFSET 4
#define PCIDEVIDOFFSET 6
#define VPDPOINTEROFFSET 8
#define PCISTRUCTLENOFFSET 0xA
#define PCISTRUCTREVISIONOFFSET 0xC
#define CLASSCODEPROGIFOFFSET 0xD
#define CLASSCODESUBOFFSET 0xE
#define CLASSCODEBASEFOFFSET 0xF
#define IMAGELENOFFSET 0x10
#define CODEREVISIONOFFSET 0x12
#define CODETYPEOFFSET 0x14
#define INDICATOROFFSET 0x15
#define RESERVEDOFFSET 0x16

#define VPDPOINTER 0x0       /* No Vital Product Data */
#define PCISTRUCTLEN 0x18   /* length of PCI data structure in bytes */
#define PCISTRUCTREVISION 0x0 /* PCI structure revision 0 - PCI spec 2.1 */
#define CODEREVISION 0x0000   /* Revision 0.0 */
#define CODETYPE 0x00        /* Intel x86 code */
#define INDICATOR 0x80       /* indicator for last ROM image */
#define RESERVED 0x0

/* Enable expansion ROM decoder.
 BestAllPropStore must be called to enable the expansion ROM decoder
at powerup.
*/
int EnableROMDecoder(b_handletype handle)
{
  b_errtype err;
  printf("Setting Board Properties ...\n");
  err = BestBoardPropSet(handle, B_BOARD_ROMUSAGE,
B_ROMUSAGE_EXTERNAL); CHECK
  err = BestPowerUpPropSet(handle, B_PU_OBSRUNMODE, 1); CHECK
  err = BestPowerUpPropSet(handle, B_PU_TRCRUNMODE, 1); CHECK
  err = BestPowerUpPropSet(handle, B_PU_CONFRESTORE, 0); CHECK
  printf("Setting up expansion ROM Decoder ...\n");
  err = BestTargetDecoderRead(handle, 7); CHECK
  err = BestTargetDecoderPropSet(handle, B_DEC_BASEADDR, 0); CHECK
  err = BestTargetDecoderPropSet(handle, B_DEC_SIZE, 18); CHECK
  err = BestTargetDecoderProg(handle, 7); CHECK
}

/*
```

```
 Open and connect to HP E2925A
 */
int OpenConnection(b_handletype *handle, b_porttype port, b_int32
portnum)
{
  if (BestOpen(handle, port, portnum) != B_E_OK)
  {
    return -1;
  }
  if (BestConnect(*handle) != B_E_OK)
  {
    BestClose(*handle);
    return -1;
  }
  return 0;
}

/* Disconnect from HP E2925A and close C-API */


void CloseConnection(b_handletype handle)
{
  BestDisconnect(handle);
  BestClose(handle);
}

/* Get size of ROM image in 512 byte blocks */
b_int32 GetROMImageSize(char *fname)
{
  struct  stat stat_buf;
  b_int32 i;

  if (stat(fname, &stat_buf) != 0)
  {
    printf("can't stat expansion rom image file\n");
    return 0;
  }       /*   file size;  offset; round up to next 512 byte multiple */
  i = (b_int32)stat_buf.st_size + EXPROMCODEOFFSET + 511UL;
  i /= 512; /* return number of 512 byte blocks */
  printf("image size: %li blocks\n", i);
  return i;
}

/* Create ROM image: reads code from the specified file, and creates
a header
 using the supplied entry point and information form the card's
configuration space. */
```

```
int CreateROMImage(
   b_handletype handle,
   char *fname,
   int imagelen512,
   unsigned int initentry,
   unsigned char *mem)
{
  int checksum=0;
  b_int32   vendor_device, classcode;
  b_errtype err;
  int romfile;
  int i;

  /* Get vendor, device and class code from card.
   These values will be used in the PCI structure.
    See example “Programming the configuration space of the E2925A”
   in the HP E2925A User’s Guide for more information on programming
    the card’s configuration space */
  err = BestConfRegGet(handle, 0x0, &vendor_device);
  err = BestConfRegGet(handle, 0x8, &classcode);


  memset(mem, 0, ROMHEADERSIZE);
  mem[ROMSIG1OFFSET] = 0x55;
  mem[ROMSIG2OFFSET] = 0xAA;
  mem[INITSIZEOFFSET] = imagelen512;
  /* 0xE9 is the near jump instruction. The next two bytes contain the
offset
  relative to the next instruction */
  mem[INITPOINTEROFFSET] = 0xE9;       /*   jmp near       */
  /* this instruction is at INITPOINTEROFFSET and occupies 3 bytes
    thus, the next instruction is at INITPOINTEROFFSET + 3
    The entry point */
  *(unsigned short*)(mem+INITPOINTEROFFSET+1) = initentry -
(INITPOINTEROFFSET + 3);
  /* Pointer to PCI structure */
  *(unsigned short *)(mem+PCISTRUCTPOINTEROFFSET) = PCISTRUCTPOINTER;
  /* PCI Structure Signature */
  mem[PCISTRUCTPOINTER+PCISTRUCTSIGOFFSET] = ‘P’;
  mem[PCISTRUCTPOINTER+PCISTRUCTSIGOFFSET+1] = ‘C’;
  mem[PCISTRUCTPOINTER+PCISTRUCTSIGOFFSET+2] = ‘I’;
  mem[PCISTRUCTPOINTER+PCISTRUCTSIGOFFSET+3] = ‘R’;
  *(b_int32*)(mem+PCISTRUCTPOINTER+PCIVENDORIDOFFSET) =
vendor_device;
  *(unsigned short *)(mem+PCISTRUCTPOINTER+VPDPOINTEROFFSET) =
VPDPOINTER;
  *(unsigned short *)(mem+PCISTRUCTPOINTER+PCISTRUCTLENOFFSET) =
PCISTRUCTLEN;
```

```
  mem[PCISTRUCTPOINTER+PCISTRUCTREVISIONOFFSET]=PCISTRUCTREVISION;
  mem[PCISTRUCTPOINTER+CLASSCODEPROGIFOFFSET]=classcode &0x00FF;
  mem[PCISTRUCTPOINTER+CLASSCODESUBOFFSET]=(classcode>>8)&0x00FF;

mem[PCISTRUCTPOINTER+CLASSCODEBASEFOFFSET]=(classcode>>16)&0x00FF;
  *(unsigned short *)(mem+PCISTRUCTPOINTER+IMAGELENOFFSET) =
imagelen512;
  *(unsigned short *)(mem+PCISTRUCTPOINTER+CODEREVISIONOFFSET) =
CODEREVISION;
  mem[PCISTRUCTPOINTER+CODETYPEOFFSET] = CODETYPE;
  mem[PCISTRUCTPOINTER+INDICATOROFFSET]=INDICATOR;

  romfile = _open(fname, _O_RDONLY | _O_BINARY);
  if(romfile == -1)
  {
    printf("Could not open file: %s\n", fname);
    return -1;
  }
  _read(romfile, mem+EXPROMCODEOFFSET, imagelen512*512-
EXPROMCODEOFFSET);
  _close(romfile);
  /* Calculate checksum */
  for(i=0; i < imagelen512*512-1; i++)
    checksum+=mem[i];
  checksum &= 0x00FF;
  mem[imagelen512*512-1]=0x0100-checksum;
}

/* Write ROM image to HP E2925A */
int WriteROMImage(b_handletype handle, int  imagelen512, unsigned
char *image)
{
  int i;
  b_errtype err;
  for(i=0; i<imagelen512*512; i++)
    /* Write one byte at offset i */
    err=BestExpRomByteWrite(handle, i, image[i]); CHECK
}

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

      int i;
  b_errtype err;
  b_handletype handle;
      b_int32 devid, byte;
  char romfile[256];
  b_int8   imagelen, imagelen512;
```

```
  unsigned char *image;
  unsigned int initentry;


  /* open the communication session to card */
  printf("Opening ...\n");
  err=OpenConnection(&handle,B_PORT_PARALLEL, B_PORT_LPT1); CHECK

  printf("Enter ROM filename:");
  gets(romfile);
  printf("Initialization code entry point:");
  gets(romfile);
  imagelen512 = GetROMImageSize(romfile);
  image = malloc(imagelen512*512);
  CreateROMImage(handle, romfile, imagelen512, initentry, image);
  fscanf(stdin, "%x", &initentry);
  printf("Programming expansion ROM ...\n");
  EnableROMDecoder(handle);
  err = BestAllPropStore(handle); CHECK
  WriteROMImage(handle, imagelen512, image);
  printf("Setting Power-Up Properties ...\n");
  err = BestAllPropStore(handle); CHECK

  printf("Closing connection ...\n");
  CloseConnection(handle);
       free(image);
  return 0;
}
```

Porting the PCI interface driver to a UNIX platform

One of the interfaces which can be used to communicate with the HP E2925A is the PCI interface. This interface uses registers in the configuration space of the card. A Windows NT/ Intel x86 driver is provided for this interface. If you would like to use the C-API on another platform you must write a device driver for it. The following steps are required:

1. Start with pci16.c
2. Implement low-level functions in a device driver. ioctl read and write functions will be implemented in our driver
3. Replace low-level functions in pci16.c with ioctl read and write function calls
4. test the device driver

C-API PCI support functions

We will start with pci16.c, which is part of the DOS version of the C-API. In this example we will write a device driver for Linux (a freely-available UNIX-like OS). Most of the information in this application should be applicable to any modern operating system.

pci16.c contains low-level functions used to access the config space of the host/DUT and to read/write data to/from the HP E2925A card. All communication with the card is done through the card's configuration space.

pci16.c defines the following functions:

- BestPCIDWSet

  Write a configuration space dword. This function is used by other functions in pci16.c

- BestPCIDWGet

  Read a configuration space dword. This function is used by other functions in pci16.c

- BestGenericPCIDWSet

  Write a configuration space dword. This function is used by other parts of the C-API.

- BestGenericPCIDWGet

  Read a configuration space dword. This function is used by other parts of the C-API.

- BestOpenPCI

Opens the PCI driver. This function sets the bus and slot number but does not communicate with the card. We will open a device file corresponding to our driver in this example.

- BestClosePCI

  Closes the PCI driver.

- BestPCIOpenConnection

  Opens a connection to the card. This function sets the connection bit in the HP E925A's configuration space.

- BestPCIReleaseConnection

  Closes the connection to the card. This function sets the connection bit in the HP E925A's configuration space.

- BestPCIWaitForClose

  Polls the connection status register of the HP E2925A and wait until the connect bit is cleared.

- BestPCIMailboxWrite

  Writes a single byte to the mailbox send register of the HP E2925A.This function waits until the mailbox send register is ready (by polling the mailbox status register) and then writes a byte to the mailbox data register. This function uses BestGenericPCIDWGet and BestGenericPCIDWSet to access the appropriate configuration space registers (mailbox data and mailbox status). We will not need to modify this function.

- BestPCIMailboxRead

  Reads a single byte from the mailbox receive register of the HP E2925A.This function waits until the mailbox receive register is ready (by polling the mailbox status register) and then reads a byte from the mailbox data register. This function uses BestGenericPCIDWGet and BestGenericPCIDWSet to access the appropriate configuration space registers (mailbox data and mailbox status). We will not need to modify this function.

- BestPCIDriver

  This function reads and writes data from/to the card.

Structure of the device driver

The device driver consists of the following functions

- best_ioctl

This function is the entry point for the functions used to connect to the card, to set up data transfers and to access the configuration space directly. The following IOCTLs are defined:

- IOCTL_BEST_GET_CONFIG_DW
  Read a configurations space dword. This is implemented in ReadConfigDW.
- IOCTL_BEST_SET_CONFIG_DW
  Write a configuration space dword. This is implemented in WriteConfigDW.
- IOCTL_BEST_SET_SLOT_NUMBER
  Set the bus number, slot number and function number of the HP E2925A to connect to. This is implemented in best_set_device() and must be called before connecting to the card. This function sets the state of the driver to slotassigned. This IOCTL will be used in BestOpenPCI.
- IOCTL_BEST_CONNECT
  Connect to the card. This IOCTL will be used in BestPCIOpenConnection. It is implemented in best_connect(). This function sets the state of the driver to connected.
- IOCTL_BEST_DISCONNECT
  Disconnect from the card. This IOCTL will be used in BestPCIReleaseConnection. It is implemented in best_disconnect().
- IOCTL_BEST_GET_CONNECT_STATUS
  Read the connection status register.
- IOCTL_BEST_GET_LAST_ERROR
  Returns the last error which occurred during a read or write.
- IOCTL_BEST_SET_REGISTER
  Set the register number and the width of the register. This register is the internal register of the HP E2925A which will be read from/written to in the next call to read()/write(). The same configuration space data register is always used irrespective of the internal register.

- best_read

  Read data from the HP E2925A card. This function has the same structure as BestPCIDriver in pci16.c. BestPCIDriver will call this function to read data from the card.

- best_write

  Write data to the HP E2925A card.

- best_open

  Open the PCI interface driver.

- best_release

  Close the PCI interface driver.

- init_module and cleanup_module

  init_module is Linux-specific and is called when the device driver is loaded. cleanup_module is also Linux-specific and is called when the device driver is unloaded.

The driver maintains an array of 8 bestdevice structures.

```
struct bestdevice devs[MAXBESTDEVICES];
```

This array is accessed using the minor number of the device as an index.

The bestdevice data structure is declared as follows:

```
struct bestdevice {
       unsigned char regno;
       unsigned char regwidth;
       unsigned char DevFn;
       unsigned char BusNumber;
       bestdriverstate state;
       unsigned char status;
};
```

- regno

  The internal register to read from/write to

- regwidth

  The width of the internal register to read from/write to

- DevFn

  The device number and function number of the HP E2925A card associated with this instance of the driver.

- BusNumber

  The bus number of the HP E2925A card associated with this instance of the driver.

- state

  The current state of the driver. The driver may be in one of the following states:

  - loaded
    All devices are in this state initially
  - open

This instance of the driver has been opened.

- slotassigned
  A bus number and slot number has been assigned to the device.
- connected
  The card has been connected to.
- registerassigned
  A register number and width have been assigned for the next data transfer to the card
- reading
  A read access is in progress.
- writing
  A write access is in progress.

- status

  The last status value returned by the card. This member is updated at the end of each block transfer.

Opening and closing the device driver

BestOpenPCI is called by BestOpen. BestOpen uses an array of b_handlestype structures to store information about the cards associated with each card. The ‘handle’ parameter used by the C-API functions is an index into this array. This structure includes a portnumber member which stores the file handle of associated device file.

```
extern b_handlestype handle_array[];
```

BestOpen passes the slot number and bus number to BestOpenPCI in portnum. BestOpenPCI first searches for an unopened device file. Device files bestpci0 - bestpci7 are associated with our driver and have minor numbers 0-7.

```
for (i = 0; i < MAXBESTDEVICES; i++) {
    sprintf(devname, “/dev/bestpci%i”, i);
    pcihandle = open(devname, O_RDWR);
    if(pcihandle != -1)
        break;
}
```

If a device file has been successfully opened, the bus number and slot number are passed to the driver using IOCTL_BEST_SET_SLOT_NUMBER.

```
/* now set up slot number in the driver */
ioctlresult=ioctl(pcihandle,IOCTL_BEST_SET_SLOT_NUMBER,&slotnum);
if (ioctlresult != 0) {
```

```
    close(pcihandle);
    pcihandle = -1;
}
```

best_ioctl calls best_set_device. This function checks that this instance of the device driver hasn't connected to a card and if not sets the device number and bus number.

```
slotnum = get_user((unsigned int *) arg);
    ...
if (devs[minor].state != open && devs[minor].state != slotassigned) {
    return -EINVAL;
}
devfn = (unsigned char) (slotnum & 0x000000FF);
busno = (unsigned char) ((slotnum >> 8) & 0x000000FF);
devs[minor].DevFn = devfn;
devs[minor].BusNumber = busno;
devs[minor].state = slotassigned;
```

BestClosePCI is called by BestClose. BestClosePCI closes the device file:

```
close((int) portnumber);
```

Connecting to and disconnecting from the HP E2925A

The software connects to and disconnects from the card by writing to the connect command register. The connect bit of the connection status register will be set by the card to acknowledge the connection and cleared when the software disconnects.

BestPCIOpenConnection in pci16.c writes to the connect command register and waits until the card has acknowledged the connection. IOCTL_BEST_CONNECT and IOCTL_BEST_GET_CONNECT_STATUS are implemented by the driver to support this function. The following is an extract from BestPCIOpenConnection in unixpci.c

```
ioctl((int) portnumber, IOCTL_BEST_CONNECT);
...
while ((timeout > 0) && ((BestPCIGetConnectionStatus((int) portnumber)
     & PCI_CONNECT_STATUS_BIT) == 0)) {
    timeout--;
}
```

The following extract is taken from BestPCIGetConnectionStatus in unixpci.c:

```
ioctl((int) portnumber, IOCTL_BEST_GET_CONNECT_STATUS, &status);
return status;
```

best_connect in bestpcid.c checks that the card has not been connected to and, if it hasn't, writes to the connect command register:

```
pcibios_write_config_dword(devs[minor].BusNumber,
    devs[minor].DevFn, PCI_CONNECT_CMD, PCI_CONNECT_CMD_BIT)
```

The following is an extract from BestPCIReleaseConnection in unixpci.c

```
ioctl((int) portnumber, IOCTL_BEST_CONNECT);
```

best_disconnect in bestpcid.c checks that the card has been connected to and, if it has, clears the connect command register:

```
pcibios_write_config_dword(devs[minor].BusNumber,
    devs[minor].DevFn, PCI_CONNECT_CMD, 0x0);
```

After calling BestPCIReleaseConnection, the C-API calls BestPCIWaitForClose. The following is an extract from BestPCIWaitForClose:

```
while ((timeout > 0) &&
    ((BestPCIGetConnectionStatus((int) portnumber)
    & PCI_CONNECT_STATUS_BIT) != 0)) {
    timeout--;
}
```

This function waits until the connection status bit is cleared by the card.

Reading and writing

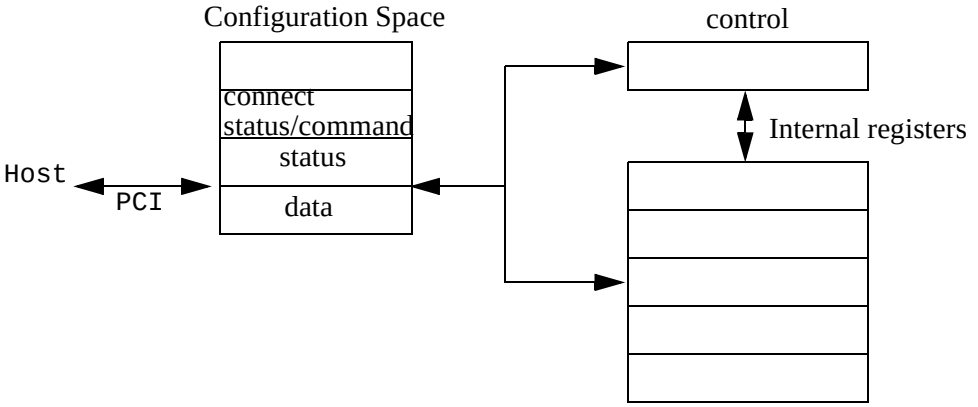
Two registers within the configuration space of the E2925A are used to communicate with it.

- Data register

  The data register is used to transfer data to/from the card.

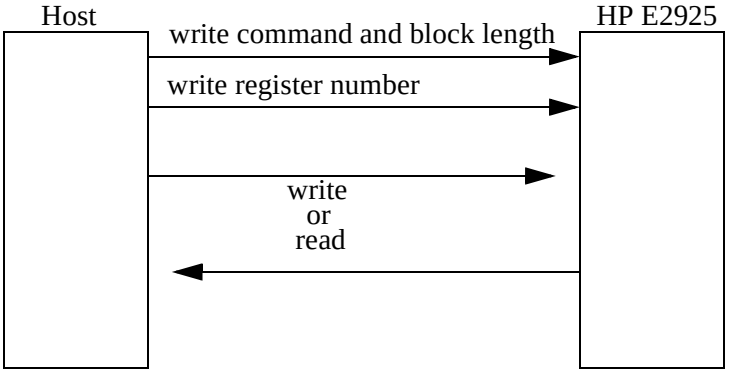
- Status register

  The status register indicates whether or not the card is ready to accept data and whether or not the card has available data.



The E2925A is programmed through the data register. The data register is 32-bit register but is accessed using byte/word/dword accesses depending on the internal register being accessed. The internal registers of the E2925A are addressed by writing the register number to the data port.

Reading/writing a block of data the card



One function in pci16.c is used to read from the card and to write to it. This function is BestPCIDriver and we will move most of the code in this function into the device driver. This function is eventually called by all API functions which use the card except the mailbox functions.

To understand how this function is used we will follow the execution of BestDummyRegWrite. This function writes to an otherwise unused register on the card. It can be used to test and debug the device driver and is used by BestConnect to check the connection to the card.

```
err = BestDummyRegWrite(handle, 0x00ff00ff);
```

The above line calls the API function, using a handle to identify the device. As in most other API functions this results in a call to BestBasicBlockWrite (other functions may also use BestBasicBlockRead). This is the function used internally by the API to communicate with the card, independently of the interface used.

```
BestBasicBlockWrite(handle, 4, &value, 4);
```

It determines the appropriate function to call for the interface associate with this handle. In the case of the PCI interface it calls BestPCIDriver.

```
BestPCIDriver(handle, 0xFF, CMD_WRITE, &data, 4, 4);
```

This function is called by both BestBasicBlockRead and BestBasicBlockWrite. It splits the data into blocks, if necessary, and reads/writes the data to the card. It writes two control bytes to the card at the beginning of a block. It then writes It waits until the card is ready. This functionality will be implemented by the device driver in this example.

The read and write functions

BestPCIDriver first sets the internal register number which will be written to/read from and the width of this register:

```
if (BestPCISetRegister(pcihandle, regwidth, addr) == FALSE)
    return B_E_ERROR;
```

The following is an extract from BestPCISetRegister:

```
bestinternalregister reg;
reg.regno = regno;
reg.regwidth = regwidth;
ioctlresult = ioctl(pcihandle, IOCTL_BEST_SET_REGISTER, &reg);
```

Reading is implemented as follows:

```
switch (cmd) {
```

```
case CMD_READ:
    /* get data from the HP E2925A */
    ReturnedLength = read(pcihandle, byteptr, len);
    if (ReturnedLength == -1) {
        status = BestPCIGetLastError(pcihandle);
        if (status == STATUS_FUNC_ERROR)
            return B_E_FUNC;
        else
            return B_E_ERROR;
    }
    break;
```

BestPCIDriver reads from the device file. If the read function returns an error BestPCIGet-LastError is used to determine the last error value read from the card.

Writing is implemented as follows (error checking not shown):

```
case CMD_WRITE:
    /* send data to the HP E2925A */
    ReturnedLength = write(pcihandle, byteptr, len);
    ...
```

read The read device driver function first checks that the register number and width have been set. It then checks that the calling process has permission has permission to write to the buffer.

```
if (devs[minor].state != registerassigned) {
    return -EINVAL;
}
devs[minor].state = reading;
...
rc = verify_area(VERIFY_WRITE, buf, count);
if (rc != 0)
    return rc;
```

best_read then calculates the number of accesses required and uses best_read_block to read data in blocks of up to PCI_MAX_LEN_PER_BLOCK(127) transfers:

```
accesscount = count / devs[minor].regwidth;
while (accesscount > 0) {
    if (accesscount > PCI_MAX_LEN_PER_BLOCK)
        hlen = PCI_MAX_LEN_PER_BLOCK;
    else
        hlen = accesscount;
    accesscount -= hlen;
    rc = best_read_block(minor, buf, hlen);
    if (rc == -1) {
```

```
            nread = -1;
            break;
        }
        nread += rc * devs[minor].regwidth;
        buf += hlen * devs[minor].regwidth;
    }
```

accesscount is the remaining number of accesses required to transfer the requested number of bytes. hlen is the number of transfers in the current block. accesscount is decremented by hlen in each iteration.

When best_read has finished reading from the card, it sets the state of this device to ‘connected’, ready for the next read/write.

```
best_set_state(minor, connected);
```

best_read_block writes a header to the card indicating the length of the block and the command (CMD_READ in the case of a read. best_write_block uses the CMD_WRITE command).

```
best_begin_block(minor, CMD_READ, hlen);
```

best_begin_block first checks the connection status and returns an error value if the card has disconnected.

```
pcibios_read_config_dword(devs[minor].BusNumber,
        devs[minor].DevFn, PCI_CONNECT_STATUS, &connect_status);
if ((connect_status & PCI_CONNECT_STATUS_BIT) == 0)
    return -1;
```

It then writes the command and block length (number of transfers) using a single configuration write. Before writing to the card it must wait until the card is ready to accept data.

```
best_write_wait(minor);
pcibios_write_config_byte(devs[minor].BusNumber,
    devs[minor].DevFn, BEST_PCI_DATA, (hlen & 0x7F) | (cmd << 7));
best_write_wait(minor);
pcibios_write_config_byte(devs[minor].BusNumber,devs[minor].DevFn,
    BEST_PCI_DATA, devs[minor].regno);
```

best_write_wait polls the status register until the send full bit is cleared:

```
do {
    pcibios_read_config_byte(devs[minor].BusNumber,
        devs[minor].DevFn, BEST_PCI_STATUS, &status);
} while ((status & PCI_STATUS_SEND_FULL_BIT) != 0);
```

When the header has been written to the card the driver reads from the card using best_read_reg.

```
for (i = 0; i < hlen; i++) {
    best_read_reg(minor, block);
    block += devs[minor].regwidth;
}
best_end_block(minor);
```

best_read_reg waits until the card is ready and reads a single value from the data register. In the case of a byte-wide register this is a configuration space byte access. When it has read the data register it clears the data valid bit in the status register.

```
best_read_wait(minor);
...
pcibios_read_config_byte(devs[minor].BusNumber, devs[minor].DevFn,
    BEST_PCI_DATA, &byte);
put_user(byte, dest);
...
best_clear_datavalid(minor);
```

best_read_wait polls the status register until the data valid bit is set.

```
do {
    rc = pcibios_read_config_byte(devs[minor].BusNumber,
    devs[minor].DevFn, BEST_PCI_STATUS, &ByteData);
} while ((ByteData & PCI_STATUS_DATA_VALID_BIT) == 0);
```

When best_read_block has finished writing a block, it calls best_end_block. best_end_block reads a single byte from the card. It waits until the card has placed the current status in the data register before reading it. After reading the status it clears the data valid bit.

```
best_read_wait(minor);
pcibios_read_config_byte(devs[minor].BusNumber, devs[minor].DevFn,
    BEST_PCI_DATA, &devs[minor].status);
pcibios_write_config_byte(devs[minor].BusNumber,devs[minor].DevFn,
    BEST_PCI_STATUS, PCI_STATUS_DATA_VALID_BIT);
```

write The write device-driver function has the same structure as the read function. It writes data to the card in blocks of up to 127 transfers using best_write_block. best_write_block writes to the card using best_write_reg. best_write_reg waits until the card is ready to accept data:

```
rc = best_write_wait(minor);
```

best_write_wait polls the PCI control interface status register and checks the send full bit. When this bit has a value of 0, the card is ready to accept data:

```
do {
    pcibios_read_config_byte(devs[minor].BusNumber,
        devs[minor].DevFn, BEST_PCI_STATUS, &ByteData);
} while ((ByteData & PCI_STATUS_SEND_FULL_BIT) != 0);
```

When the card is ready best_write_reg writes to the card's data register in its configuration space. The following extract shows the code used to write to byte-wide registers:

```
byte = get_user(src);
pcibios_write_config_byte(devs[minor].BusNumber,
    devs[minor].DevFn,BEST_PCI_DATA, byte);
```

Testing and debugging the device driver

Liberal use was made of printk (the Linux kernel equivalent of printf) during the development of this driver. Ensure that the correct IOCTL functions are called and that data is transferred correctly to/from the driver.

The E2925A may be controlled from an external interface and used to observe configuration space accesses to the card. The following should be checked for:

- The driver must write a header to the card before transferring a block (both reads and writes).
- The driver must read the status (single byte config read of the data register) after transferring a block. A non-zero value should result in a "Functional onboard error" message from the C-API.
- The driver must wait until the card is ready before it reads from or writes to the data register. This can be observed as a series of config reads from the status register, where the appropriate bit is set or cleared (see *read on page 103* and *write on page 105*).
- The driver must clear the data valid bit of the status register. This is done by writing to the status register.

The BestDummyRegRead and BestDummyRegWrite functions may be used to test the driver. They produce the simplest possible sequence of transfers. The read (write) a single dword register on the card. This involves reading (writing) a single block from the card. This block consists of one dword transfer.

You may find it useful to compile a debug version of the C-API under DOS/windows and compare the results.

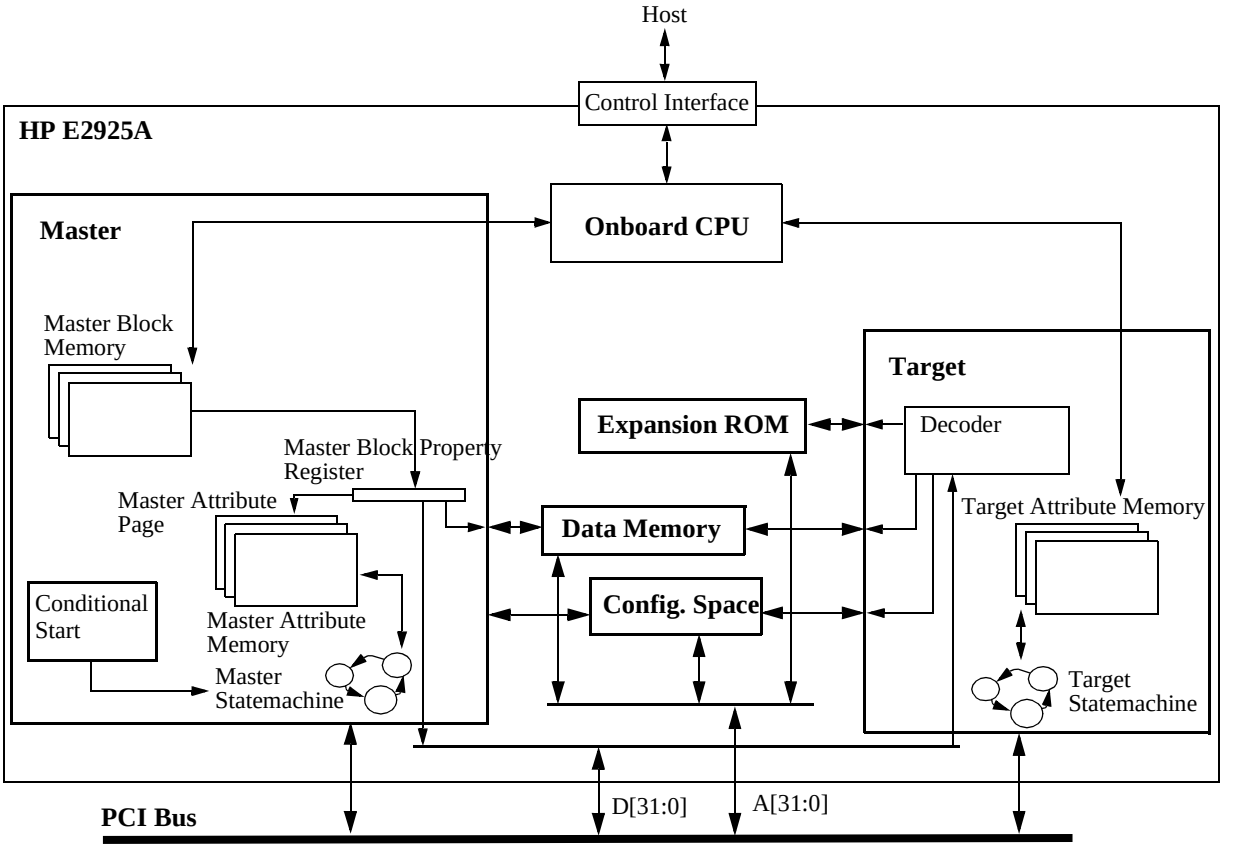
The complete source code for the device driver and test programs are available on the installation CD in the directory samples\capi\unixport. You will need to copy the C-API soources into the samples\capi\unixport\src directory, copy the header files into samples\capi\unixport\include and apply the patch b_pci_h.dif (i.e. patch < b_pci_h.dif)., if you would like to compile this example.

Chapter 5 PCI Exerciser Overview

This chapter gives a functionality overview of the of the exerciser part of the E2925A. This chapter contains the following sections:

Hardware Overview

PCI Exerciser Runtime Hardware Overview



Master Block Memory This is part of the CPU RAM and stores the master block transfer properties. The CPU reads the contents of the Master Block Memory during the master block page run and sets the Master Block Property Registers.

Master Block Property Registers This contain the block properties of the current block transfer.

Master Attribute Memory Each memory location represents the protocol attribute set of one bus phase. The Master Attribute Memory is sequenced by the master itself. Master Attribute Memory is arranged in pages.

Master Statemachine Handles a block transfer with a protocol behavior specified by the attribute page.

Master Conditional start Delays the master block transfer until, a pattern term generated in the analyzer is true and **either** a programmable delay counter, clocked with the PCI clock expires **or** a programmable timer in the CPU expires.

Data Path Handles Read and Write accesses to/from the PCI Bus.

Internal Data Memory Shared resource for master and target accesses and can be setup as memory space, memory or I/O space and as a master data buffer.

Decoders Five independent decoders with programmable size and can be individually enabled/disabled:

- memory decoder
- memory or I/O decoder
- I/O decoder for the PCI programming registers
- Expansion ROM decoder
- Configuration Space

Target Attribute Memory Each address location contains a set of protocol attributes for one address or data phase. This memory is also organized into pages.

Target Statemachine Handles target accesses with the protocol behavior of the currently activated attribute page.

One branch of the target statemachine handles configuration accesses.

PCI configuration space (programmable behavior) As defined by PCI specification 2.1

PCI Expansion EEPROM 64k Byte onboard device ROM

CPU Handles the programming accesses from the external host, and runs linear block sequences.

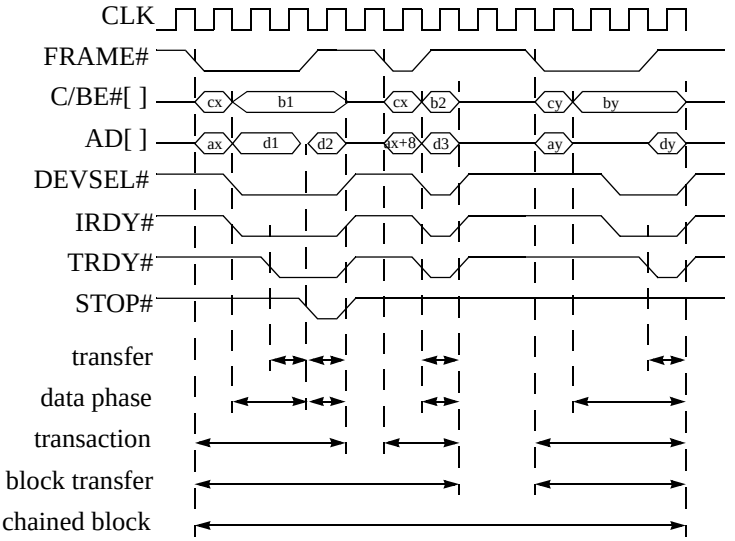
Master Operation

Hierarchical Run Concept

The PCI master implements a hierarchical run concept, where a block transfer is the basic programming construct. The protocol behavior used during the block transfer is defined by an attribute memory page.

Table 1 Hierarchical Run Concept

PCI Level	Definition
Clock Cycle	smallest granularity of PCI protocol
Data Transfer	! IRDY# & ! TRDY# (real data transfer)
Data Phase	consists of one transfer and the waits before it
Transaction	An address phase + 1 or more data phases.
Block Transfer	1 or more number of transactions, from and to a linear address space
Block Page	a linear sequence of block transfers - chained blocks



A series of blocks can be executed by defining the blocks consecutively within a block page. This run level is called chained blocks. The highest level of run abstraction is to run a built-in test function.

Table 2 Run Levels

Run Level	Properties	Performed by	C-API function to start this level
Block Transfer	busaddr, intaddr, buscmd, byten, nodwords, attrpage	HW	*BestMasterBlockRun ( ) on page 171*
Chained Blocks	block page	SW	*BestMasterBlockPageRun ( ) on page 172*
Test	chained blocks	SW	See “High Level Test Functions” on page 290.

Each Run Level may be programmed to run once, or can be looped.

Master Block Transfer

A master block transfer is the basic programming level for creating traffic on the bus. It allows one or more transactions to or from a linear address space in real-time. A block transfer might need one or more transactions to complete, depending on the intended burst length and on target disconnects and retries.

Block Properties Master block properties remain constant for a complete block transfer. The following table lists the master block properties

Table 3 Master Block Transfer Properties

Property	Value	Description
cmd	0000\b to 1111\b	PCI bus command
addr	32 Bit	PCI bus address
nofdwords	1 to 32k	Number of transfers. The number of transferred bytes depend on the byte enables
byten	0000\b to 1111\b	PCI byte enables.
int_addr	00000\h to 1FFFC\h	Byte address offset of the BEST internal data memory. Only dword addresses are allowed.
attr_page	0 to 255	Page number of the master attribute page used for the block transfer
comp_flag	0(default) /1	When set, compare nofdwords data at int_addr with data at comp_offset
comp_offset	00000 to 1FFFC\h	Location of the compare data in the internal memory. As with int_addr this address is dword aligned.

Master Chained Blocks

Master chained blocks provide a means to execute a linear sequence of block transfers in a very fast sequence, thus being able to perform fast write/read of data blocks

The complete Master Block memory contains 16 pages, each with 16 possible block transfer entries. Pages are not limited to 16 blocks and may be programmed up to the size of master block memory if desired. A maximum of 256 master blocks is therefore possible.

The CPU executes each block within a blockpage in sequence. While a block is executed by the master, the CPU prepares the block property registers in the background. The latency between blocks therefore depends upon the numbers of transactions in the previous block.

Compare Utility

The onboard CPU provides a utility which compares two data blocks within the internal memory. This utility is controlled using the block properties

- internal address of the current block
- nofdwords are compared
- compare flag (yes/no)
- compare offset, which is the internal address of the compare block.

The results of a compare is flagged by the data compare error bit in the BestStatusRegister. This utility can be used to make data level tests with built-in checks.

Master Programming

The master may be programmed using either:

- PCI Exerciser Graphical User Interface (HP E2971A)
- C-API Interface
- Command Line Interface

Graphical User Interface Programming

Master bus transactions are programmed from the HP E2971A GUI using the Bus Transaction Language (BTL) editor. The BTL consists of several bus commands which are combined to create complete transactions. Protocol behavior is controlled by parameters in the bus commands.

For example, a burst of 3 data transfers to video memory:

```
send_video_data {
        m_xact (addr=B8000\h, cmd=m_write);
        m_data (data=86458642\h);
        m_data (data=86458642\h);
        m_last (data=86458642\h);
}
```

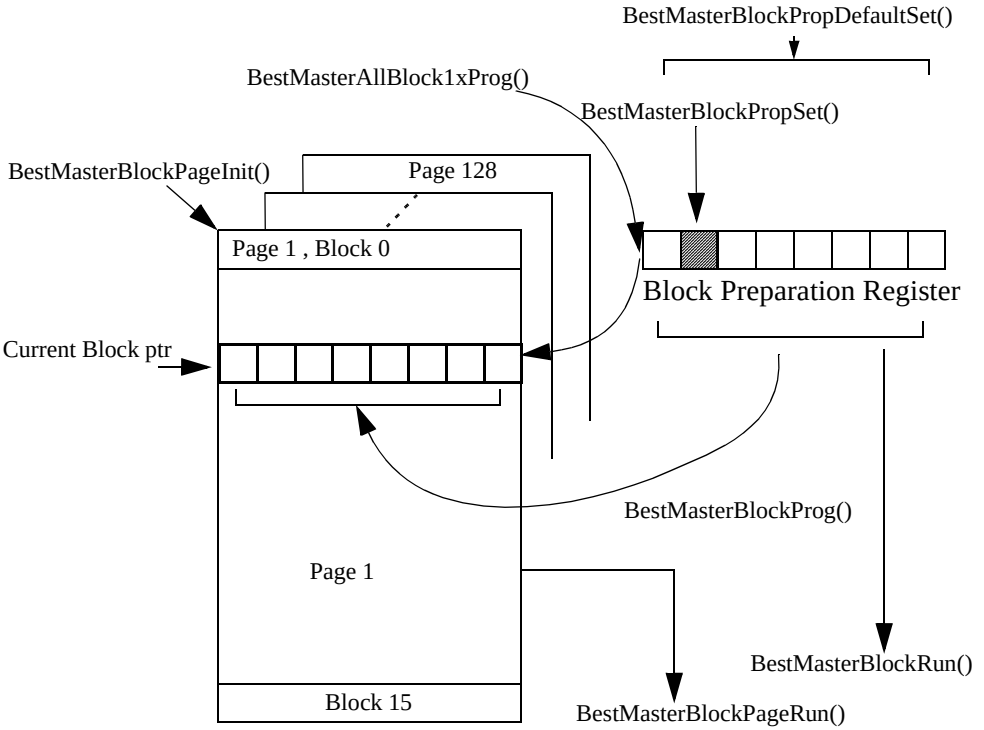
C-API Master Programming Model

From the C-API, the master is programmed by setting up master transaction blocks. A master transaction block is defined as the transfer of a block of data to and from a linear address space. The basic properties of a master transaction block are therefore: bus command, BEST internal data memory address, PCI bus address, number of transfers, byte enable and protocol behavior attribute page. The protocol attributes for each transaction block are defined by master attributes. These can be programmed on a per phase basis.

For more information of available C-API master programming functions see:
Master Programming Functions on page 290

For more information on programming master transactions using the C-API see:
Creating Master Transactions on page 46

Master Programming Model Showing C-API Functions

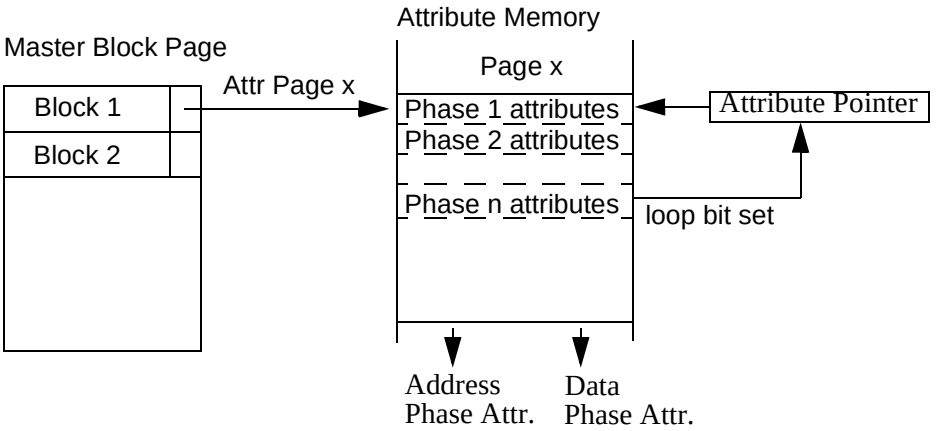


Master Block Programming Memory Map

Block page programming is organised in 128 pages with 16 blocks each. Page 0 is the default block used by internal functions. It cannot be programmed by the user. Block pages can be concatenated.

Master Protocol Attributes

Master protocol attributes provide control over the protocol used to execute a block transfer. The attribute memory is organized into 256 pages of 32 lines or phases. Each master attribute line corresponds to a data or address phase. These protocol attributes are associated to a block transfer using the attribute page number property in the master block.



Each master attribute memory location contains the control bits for one data and address phase. If the phase happens to be an address phase then the address attributes are used, the corresponding data phase attributes are then used for the first data transfer of the transaction.

Future exerciser hardware will have a maximum of 256 lines or phases each for master and target attributes, with mechanisms for repeating/ looping phases to maximize memory usage. If you wish to maintain compatibility with future exerciser hardware you should not use more than 256 protocol attribute phases/lines.

What happens during a block execution •

- The CPU starts the data path unit to prepare address, command and data pipeline for the transaction.
- It starts the master statemachine, which loads the attribute pointer with the attribute page start address.
- As soon as the master gets the data path ready signal, it pulls the REQ# line and after getting GNT# asserted it starts the transaction. It interprets the attribute bits and does the first data phase with the specified attributes.
- As soon as the first data has been transferred with IRDY#== 0 and TRDY# == 0, the attribute pointer

is incremented by one, so that it is pointing to the attribute bits of the next dataphase.

- If the master detects a 'last' bit set, it will release FRAME#, indicating that this will be the last data phase of the burst. The next phase then automatically starts with an address phase.
- If the attribute pointer sees the loop bit set, it reloads the page start address again, thus looping the structure.

Table 4 Master Address Phase Attributes

Protocol Attribute	Values	Description
aperr	0 (default) / 1	Forces SERR# active, to simulate a reported wrong parity in the address phase.
awrpar	0 (default) / 1	Inverts the PAR signal for the address phase, forcing wrong parity
stepmode	stable (default) toggle	- Normal address phase - Performs 4 address steps, with toggling address values
lock	no (default)	Normal access
	lock	Try an exclusive access
	hide_lock	Keeps LOCK# asserted during the address phase, in order to simulate an access to a previously locked target from a different master.

Table 5 Master Data Phase Attributes

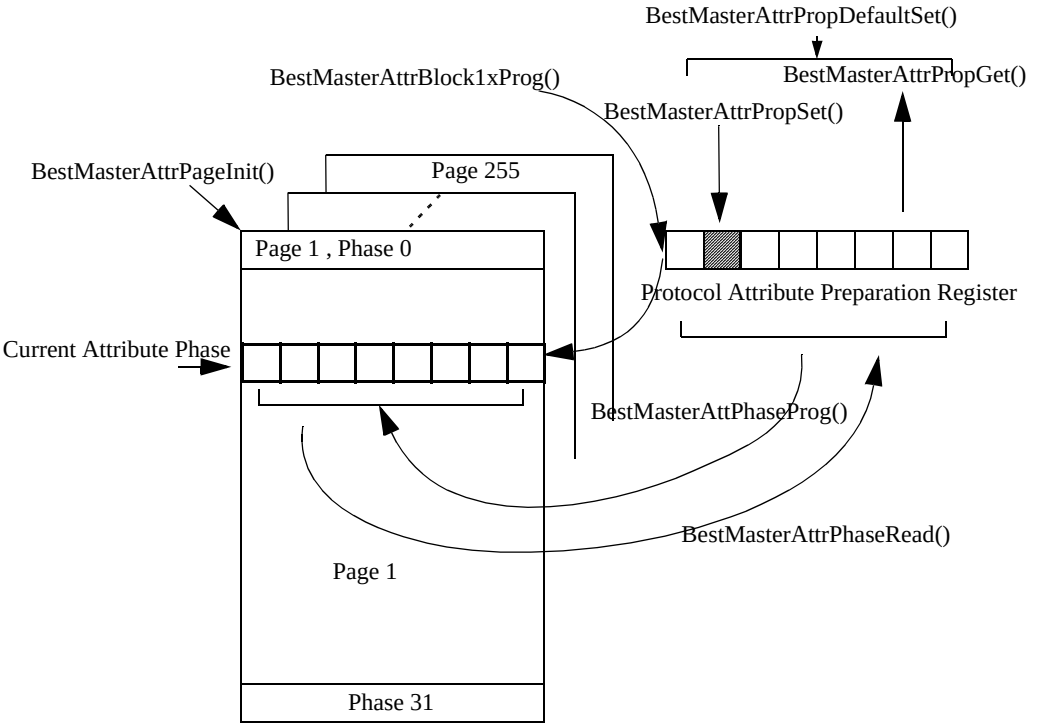
Protocol Attribute	Values	Description
waits	0 (default) to 31	numbers of waits in the data phase
dperr	0 (default) /1	Forces PERR# active, to simulate a reported wrong parity in the data phase.
dserr	0 (default) / 1	Forces SERR# active.
dwrpar	0 (default) / 1	Inverts the PAR signal for the data phase, forcing wrong parity
waitmode	stable (default) toggle	- Normal data phase, where waits are determined by the waits attribute - Performs 4 data steps, with toggling data value
last	0 (default) / 1	Forces the master to end the burst

rel_req	0 (default) / 1	Releases REQ#
do_loop	0 (default) / 1	The attribute structure will start from the page beginning in the next phase

Master protocol attributes are programmed using:

- Bus Transaction Language command parameters (HP E2971A)
- C-API Master Protocol Behavior functions

Master Protocol Attribute Programming Model Showing C-API Functions



Master Latency Timer

The master latency timer is used to ensure the BEST master cannot hold the bus indefinitely. The latency timer starts counting down (clocks) from the assertion of FRAME#, if the counter gets to zero **and** GNT# has been taken away, the master must release the bus.

The latency timer value is stored in Config Space offset 0D\h. The latency timer properties are programmed using:

- Exerciser>PCI Config>Detail in the exerciser GUI
- C-API function *BestMasterGenPropSet ( ) on page 185*

Table 6 Latency Counter

Property	Values	Description
mode	on / off (default)	Switches the latency counter on/off
counter	8Bit	

Master Conditional Start

Master conditional start is used to delay the programmed master action run until a specified pattern appears on the PCI bus.

This feature can be used to synchronize master actions to bus events.

For detailed description of pattern terms please refer to the PCI analyzer chapter (Trigger input can also be used as a pattern)

Master conditional start is programmed using:

- Exerciser>Master Cond Start in the Exerciser GUI (HP E2971A)
- C-API function *BestMasterGenPropSet ( ) on page 185*

Table 7 Master Conditional Start Run Mode Values

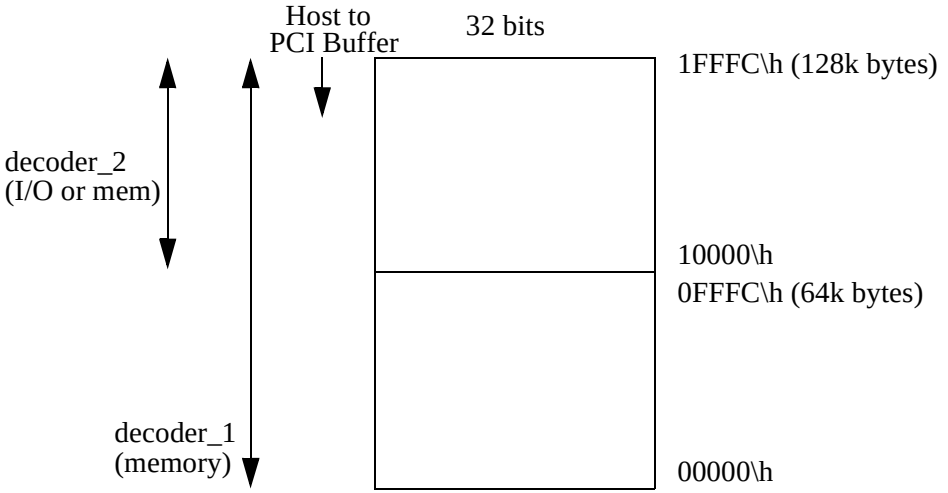
Property	Values	Description
runmode	immediate (default)	Start master action unconditionally
	wait_on_ctr	Start master after conditional start pattern occurs

Target Programming

Decoders and Internal Data Memory Model

The onboard 128k byte memory can be used for the following:

- master data buffer
- target resource for the memory decoders
- target resource for the programmable memory or I/O decoder



The host to PCI download buffer is an area of memory that can be reserved for host to PCI system memory functions. See “Host to PCI Access Functions” on page 297.

Table 8 Decoders & Associated Memory Resources

Decoder	Size (Bytes)	Address Space	Base Address	Internal Memory Address	Protocol Behavior	decode speed
1	4k to 16 M	memory	Base Address Register 0 (config space)	00000\h	Determined by target attributes	medium or slow
2	64 to 64k	memory	Base Address Register 1 (config space)	10000\h	Determined by target attributes	medium or slow
	16 to 64k	I/O	as above	as above	as above	as above
3	32	I/O	Base Address Register 2 (config space)	programming and mailbox registers	fixed, disconnect in each cycle	medium
7	256k	mem	Expansion ROM Base Address Register (config space)	Expansion ROM	fixed, disconnect in each cycle	medium
8	256	config	IDSEL	Configuration Space Header & user config space	fixed, disconnect in each cycle	medium

Decoder 1 can be set-up to decode the complete 128k. The PCI address is defined by base address register 0 (configuration space offset 10\h). If decoder_1 is programmed to a size larger than 128k, the memory will be mirrored.

Decoder 2 can be set-up to decode the top 64k bytes. The PCI address is defined by base address register 1 (configuration space offset 14\h). Note that this memory resource starts in the middle of the card internal memory (offset 10000\h).

Decoder 3 decodes 32 bytes of IO address space for accessing mailbox and onboard programming registers. This decoder is used when using the PCI bus as a controlling interface, using the DUT as the host computer. The PCI address is defined by base address register 2 (configuration space offset 18\h). For more information on Programming Register layout please refer to "PCI Controlling Interface" on page 305.. The mailbox and programming registers are already accessible through the configuration space. Decoder 3 provides another access mechanism through I/O space.

Decoder 7 decodes the onboard expansion EEPROM (or Device ROM). The Device ROM may be used as Code ROM in validation platforms. The PCI address is defined by the Expansion ROM base address register (configuration space offset 30\h). The decoder size of 256kBytes is fixed. A 64KByte EEPROM is used and memory accesses above 64K are mirrored

NOTE: The upper 1Kbyte is used for internal purposes and should not be used.

Decoder 8 decodes Configuration Space accesses.

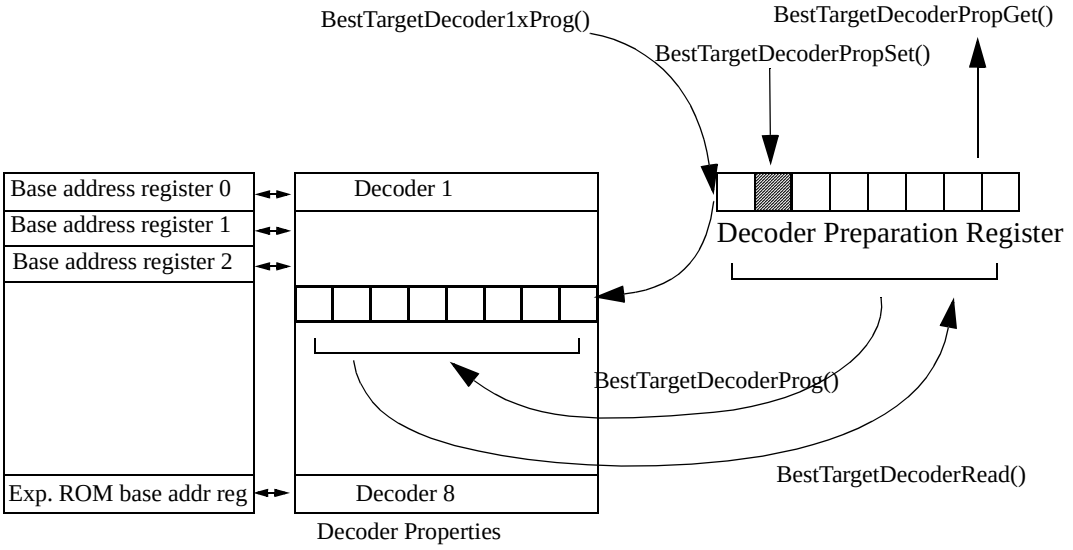
When the decoder is enabled, the size of the decoded address space is used to set the RO/RW programming mask of the corresponding Base Address Register. For example, when decoder 3 is enabled, the first 5 bits are set to RO automatically. Then during system configuration, the system can then automatically determine the size of the address space for this decoder.

NOTE: It is up to the user to write a valid start address in the base address register. This address must be on a boundary consistent with the size of the decoder.

The decoders are programmed using:

- Exerciser>Target Address Map... in the Exerciser GUI
- C-API function *BestTargetDecoderPropSet ( ) on page 194*

Target Decoder Programming Model Showing C-API Functions



The *BestTargetDecoderProg ( )* function checks that the property values in the target decoder preparation register are consistent with the specified decoder. Consistency is ensured between decoder settings and corresponding bits in the command register and the base address registers of the configuration space.

Target Protocol Attributes

Target protocol attributes provide control over the targets protocol behavior with a dataphase granularity. Target protocol attributes are programmed the same way as master protocol attributes. Like master attribute memory, the target attribute memory is organized into 256 pages of 32 lines or phases, where each line corresponds to a target address or data phase. Once the attribute page is programmed, the page must be activated. Then all accesses decoded through decoder 1 or decoder 2, will use the activated attribute page for protocol behavior.

The attribute page can be programmed from the C-API or from the user interface (HP E2971A). If using the user interface then this is done using the target attributes editor.

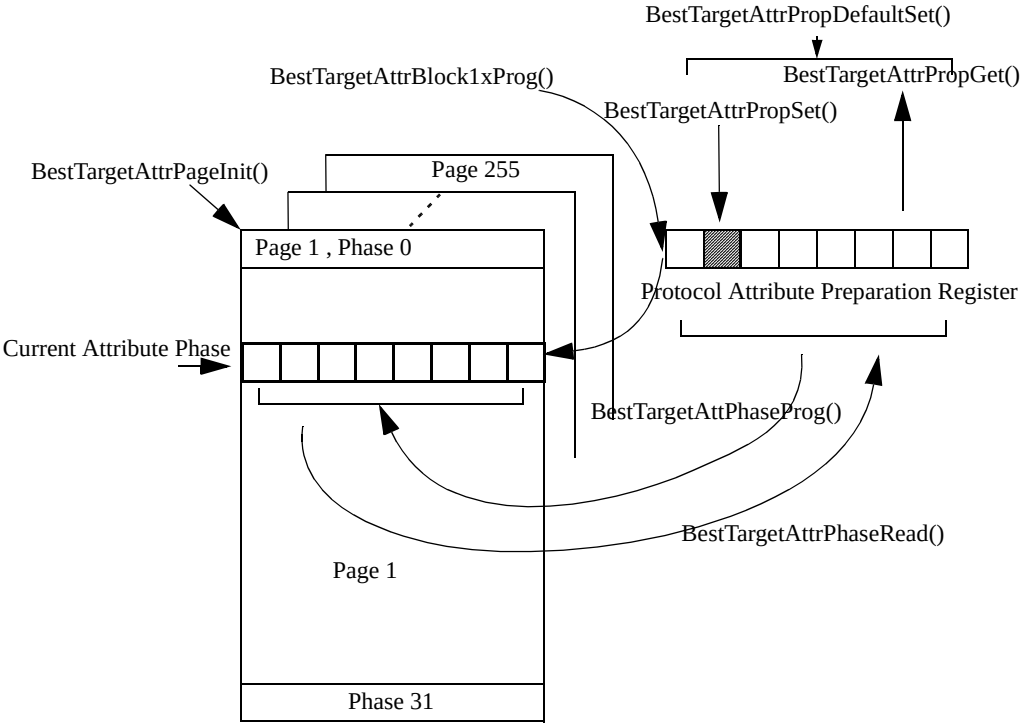
Table 9 Attribute Modes

Property	Values	Description
attr_mode	sequential	Sequential execution of target attributes. Leads to random target behavior with respect to the master accesses.
	transaction	Every address phase restarts the target attribute page. With this, target behavior is deterministic.

Table 10 Data Phase Protocol Attributes

Protocol Attribute	Values	Description
waits	0 (default) to 31	For read and write cycles except for the first phase of a read cycle, where the minimum is 2 waits
d_perr	0 (default) / 1	forces PERR# to be set, in order to simulate a reported wrong parity in the data phase.
d_serr	0 (default) / 1	forces SERR# active
d_wrpar	0 (default) / 1	inverts the PAR signals for the data phase
term	noterm, retry disconnect, abort	target termination type
do_loop	0 (default) / 1	The attribute structure repeats from the beginning of the page after a phase where the loop bit is set.

Target Protocol Attribute Programming Model Showing C-API Functions



Configuration Space

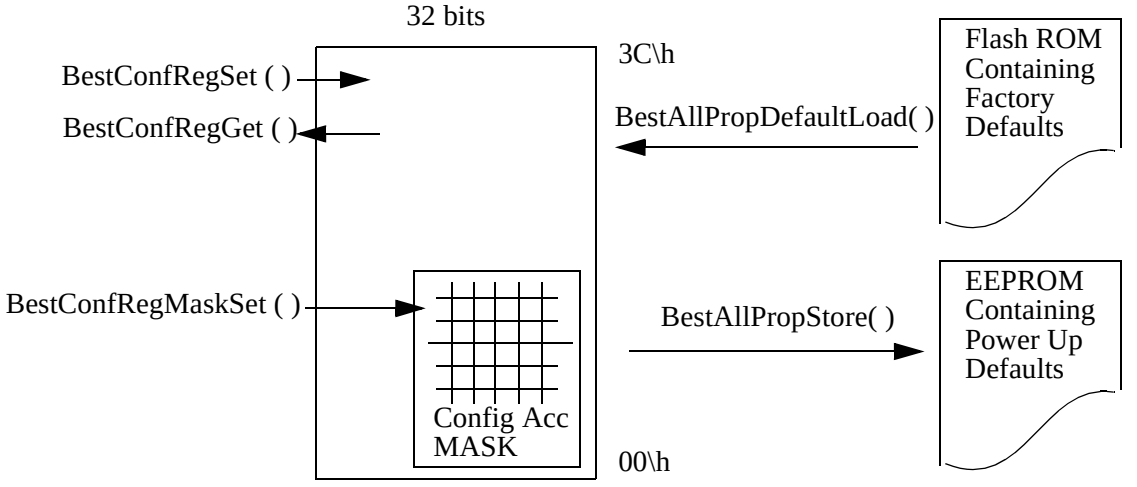
The exerciser provides a fully programmable PCI Configuration Space. This enables the card to behave like a real single function PCI device with real configuration space. For example, if the exerciser target does a target abort, then the corresponding bit in configuration space is set.

Factory default values are stored in the onboard CPU Flash ROM. The power up defaults are stored in the Expansion EEPROM, and can be reprogrammed.

The configuration space can also be disabled so the card is invisible to BIOS or system configuration routines.

A programming mask is used to define which bits in configuration space are programmable through configuration accesses (RW - Read/Write) and which are not programmable (RO - Read Only). The programming mask is therefore identical in size to the configuration space and is itself fully programmable.

Hint: All Configuration Space Registers are programmable from the controlling interface, independent of the Programming Mask.



The following abbreviations are used in Table 11:

RO	Read Only
RC	Read/Clear. Register is cleared when written to.
RW	Read/Write

Table 11 Configuration Space Header Default Values and Default Programming Mask

Register [offset]	Default Programming Mask	Factory Default Value	Description
Vendor ID [0, 1]	RO	103C\h	HP's Vendor ID, 16 bits
Device ID [2, 3]	RO	2925\h	The product number of the card
Command [4, 5]	RO/RW	0000\h	See "Command Register Defaults" on page 131.

Status [6, 7]	RO/RC		See “Status Register Defaults” on page 132.
Revision ID [8]	RO	0	Device specific revision identifier
Class Code [9-0B]	RO	00\h	(offset 09) programming interface (unused)
	RO	00\h	(offset 0A) sub-class (unused)
	RO	FF\h	(offset 0B) base class. BEST does not fit an existing class code
Cache Line Size [0C]	RW	0	Supported sizes are 0, 4, 8 dwords. This information is used when the master generates MWI cycles with cacheline wrap mode. The target does not use this information.
Latency Timer [0D]	RW	00\h	Holds the master latency timer value (in PCI clocks), “Master Latency Timer” on page 120.
Header Type [0E]	RO	00\h	Header Type. BEST is a single function type device with header type 00\h
BIST [0F]	RO	00\h	Not implemented
Base Addr Registers 0 to 5 [10 - 27]	RO/RW	0	Base Addr Reg 0 [10\h)
CIS Pointer [28]	RO	0	Not used
Subsys Vendor ID [2C]	RO	0	Not used
Subsystem ID [2E]	RO	0	This can be used to distinguish between several BESTs
Exp. ROM BAR [30]	RO	0	Enabled via the expansion ROM decoder
Reserved 0 [34]	RO	0000\h	
Reserved 1 [38]	RO	0000\h	
Interrupt Line [3C]	RW	00\h	Defines the currently used interrupts
Interrupt Pin [3D]	RO	01\h (INTA#)	The card is capable of asserting all 4 interrupt lines, but may only signal INTA# in the Status Register.
MIN_GNT [3E]	RO	00\h	
MAX_LAT [3F]	RO	00\h	

Command Register Defaults

Hint: For bits labelled RC - Doing a Config Write (value 1) to this bit clears it

Table 12 Command Register Offset 04-05

Bit	Default Programming Mask	Default Value	Description
0	RW	0	IO Space Control
1	RW	0	Memory Space
2	RW	0	Bus Master Control
3	RO	0	Special Cycles (not capable to respond to Spec Cycles)
4	RW	0	Memory Write and Invalidate Enable
5	RO	0	VGA Pallet Snoop (not snoop capable)
6	RW	0	Parity Error Response
7	RO	0	Wait cycle Control
8	RW	0	SERR# Enable
9	RO	0	Master fast back to back enabled
15:10	RO	0	Reserved

Status Register Defaults

Table 13 Status Register Offset 06-07

Bit	Default Programming Mask	Default Value	Description
5	RO	0	66 MHz capable (default = no)
6	RO	0	UDF supported (default - no)
7	RO	1	Target fast back to back capable (default - yes)
8	RC	0	Data Parity Error Detected
9,10	RO	01	Decode Speed (default - medium)
11	RC	0	Signalled Target Abort
12	RC	0	Received Target Abort
13	RC	0	Received Master Abort
14	RC	0	Signalled System Error
15	RC	0	Parity Error detected

Base Address Register Defaults

Base address registers 0, 1, and 2, are used by target decoders 1, 2, and 3 respectively.

The values contained in the Base Address Registers is controlled from the Target Decoder Setup. This can be modified from either the Exerciser GUI, or through the C-API. Factory defaults have decoders 1 (memory), 2 (IO), and 8 (config) enabled.

For more information on target decoders, see
"Decoders and Internal Data Memory Model" on page 122.

Table 14 Base Address Register [0], Offset 10\h, Decoder 1

Bit	Default Programming Mask	Default Value	Description
0	RO	0	Address Space (default - memory space)
[2:1]	RO	00	Memory Space Location (default - 32 bit address space)
3	RO	1	Prefetchable
[11:4]	RO	0	When decoder 1 is enabled (default), the bits which are RO are dependent on the “size” property for this decoder (default size is 4096 bytes). If the decoder is disabled all bits are Read Only (RO), “BestTargetDecoderPropSet ( )” on page 194.
[31:12]	RW	0	Default address range for decoder 1 is 4096 bytes.

Table 15 Base Address Register [1], Offset 14\h, Decoder 2

Bit	Default Programming Mask	Default Value	Description
0	RO	1	Address Space (default - IO space)
1	RO	00	Reserved
[3:2]	RO	0	Default address range for decoder 2 is 16 bytes.
[31:4]	RW	0	Default address range for decoder 2 is 16 bytes.

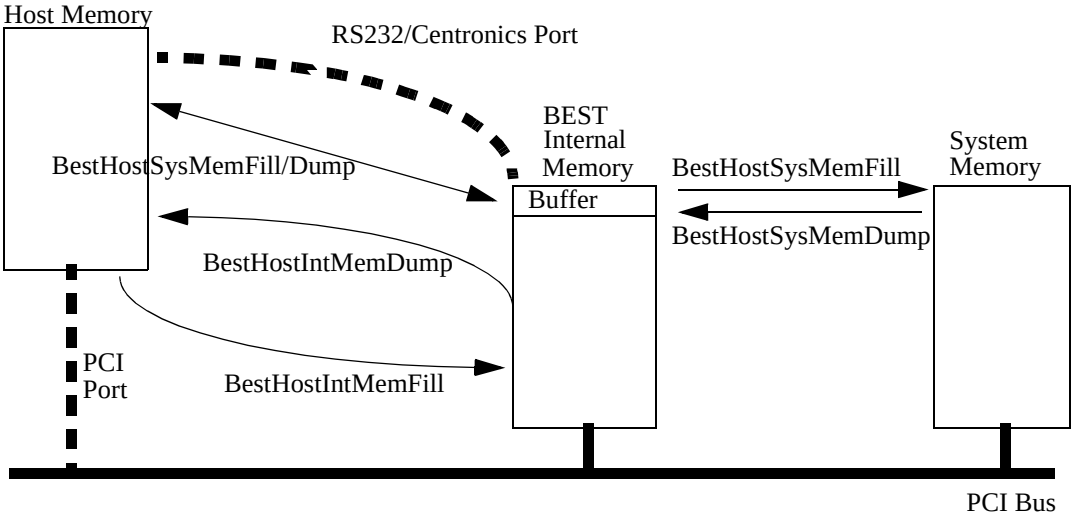
Table 16 Base Address Register [2], Offset 18\h, Decoders 3

Bit	Default Programming Mask	Default Value	Description
[31:0]	RO	0	Decoder 3 is disabled by default.

Table 17 Base Address Registers [3], [4], [5], Offset 1C\h, 20\h, 24\h

Bit	Default Programming Mask	Default Value	Description
[31:0]	RO	0	General Purpose Base Address Registers

Host to/from PCI System Memory



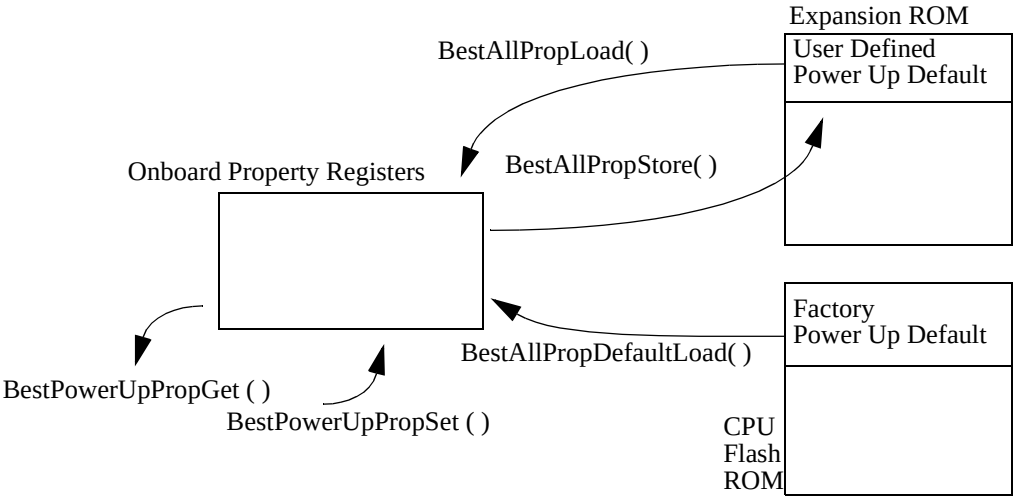
These functions can be used to download code to the DUT. For more information see “Host to PCI Access Functions” on page 297.

Interrupt Generator

Interrupts can be programmed using C-API function *BestInterruptGenerate ( ) on page 255*. All interrupts are signalled in the Best Status Register. The interrupts can be cleared by clearing the corresponding status bits in the BestStatusRegister.

Power-Up Behavior

Immediately after power-up the onboard CPU executes *BestAllPropLoad ( )*, which reloads the user defined power-up default properties to the onboard property registers.



To create your own power-up properties from the C-API:

- Use the xxxPropSet function to set each property. For an overview of the functions which affect each property see “Overview of Programming Functions” on page 290.
- then use the *BestAllPropStore ( )* function to store the properties as power-up defaults in Expansion ROM.

To enable the analyzer on powerup use *BestPowerUpPropSet ( )*.

To create your own power-up properties from the GUI

- Ensure that the windows reflect the power up state you want to store.
- then use the “File>Save as Power Up Defaults” menu option to store the properties as power-up defaults in the Expansion ROM.

System Reset

A system reset can be initiated from either the C-API, from the GUI or from RST# on the PCI bus.

BEST Board Reset

This reset is equivalent to re-powering the card or pressing the hard reset button on the board. It resets all statemachines, the target decoders and configuration space. The board reinitializes with the user defaults stored in the onboard EEPROM

See also “Power-Up Behavior” on page 136

Programmatically, this reset can be initiated using the *BestBoardReset ( )* C-API function.

BEST Statemachine Reset

This resets the master and target statemachines. The target decoders and memories are not affected.

This reset is initiated programmatically using the *BestSMReset ( )* C-API function.

PCI Reset

A PCI reset is activated from the PCI RST# over the PCI bus. There are 2 PCI reset modes:

Statemachine Reset Mode Equivalent to BEST Statemachine Reset, see above.

Reset All Equivalent to BEST Board Reset, see above.

From the C-API, the PCI Reset mode is set using function *BestBoardPropSet ( ) on page 275*

Chapter 6 PCI Analyzer Overview

This chapter gives an overview of the functions and features of the PCI analyzer.

This chapter contains the following sections:

Analyzer Overview

Protocol Observer

The protocol observer monitors 25 protocol rules in real-time. Each rule can be individually suppressed using a bit mask to disable the detection of known problems. As well as providing an “any error” output for triggering purposes, registers are used to latch the first error to occur and accumulate errors. The protocol observer status can be displayed on the on-board hex display.

State Analyzer Trace Memory

The trace memory stores all PCI signals, the exerciser state, bus state and additional information for cross referencing with the data listers. The analyzer provides 32K deep trace memory.

Trigger& Storage Qualifier

2 pattern terms are able to compare the input signals to a 1/0/X pattern. One pattern term is a dedicated trigger for the state analyzer trace memory, the other serves as a storage qualifier. The trigger point is always in the centre of the 32 K trace memory. This means you always capture 16 K samples prior to the trigger point and 16 K samples after.

External trigger

The analyzer may be triggered from an external source using up to 4 inputs connected to the trigger I/O connector on the Main Board.

Heartbeat Trigger

The heartbeat trigger is used to trigger the analyzer if an event does not occur within a defined period. This allows you to setup a pattern (for example, FRAME# going low, or an IO write), and a duration in PCI clock cycles. The analyzer will then trigger if the pattern does not occur within the duration. The trigger is enabled after the first occurrence of the pattern.

Optional Logic Analyzer Connection

Option 003 provides an adapter which can be used to connect an HP logic analyzer.

Powerup behaviour of the analyzer can be specified

Protocol Observer

The protocol observer checks 25 protocol rules in real-time. Each rule can be individually masked to disable detection of known problems. The protocol observer may provide trigger information for the trace memory. The content of the error registers can be read out from the C-API. The first error is displayed in *hexadecimal* on the LED display, if it is in default mode (see *BestDisplayPropSet ( ) on page 260* for information on setting the hex display mode).

The following protocol rules are checked:

Rule	Rule Name	Description
0	frame_0	FRAME# must be deasserted as soon as IRDY# can be asserted, whenever STOP# is asserted (PCI Spec. Appendix C, Rule 12 e)
1	frame_1	Fast-Back-to-Back is only allowed after a write- transaction or master abort (PCI Spec. Sect. 3.4.2. Fast Back-to-Back Transaction)
2	irdy_0	IRDY# must not be asserted on the same clock edge that FRAME# is asserted but one or more clocks later (PCI Spec. Sect. 3.3.1. Read Transaction - see timing diagram)
3	irdy_1	FRAME# cannot be deasserted unless IRDY# is asserted (PCI Spec. Appendix C, Rule 8 c)
4	irdy_2	IRDY# must be deasserted after last transfer or when FRAME# is high and STOP# was asserted (PCI Spec. Sect. 3.3.3.1. and 3.3.3.2.1, Rule 1)
5	irdy_3	a transaction starts when FRAME# is sampled asserted for the first time => IRDY# must not go low when FRAME# is high (PCI Spec. Appendix C, Rule 7)
6	irdy_4	once a master has asserted IRDY# it cannot change IRDY# or FRAME# until the current data phase completes (PCI Spec. Appendix C, Rule 8 d)
7	devsel_0	DEVSEL# must not be asserted during special cycle or if a reserved command was used (PCI Spec. Sect. 3.7.2. Special Cycle and Sect. 3.1.1. Command Definition)
8	devsel_1	DEVSEL# must not be asserted when FRAME# is high or was sampled high on the last clock edge (for DAC DEVSEL# must be delayed for one cycle). (PCI Spec. Sect.3.3.1. Read Transaction - see timing diagram)
9	devsel_2	once DEVSEL# has be asserted, it cannot be deasserted until the last data phase has completed (except to the signal target-abort) (PCI Spec. Appendix C, Rule 35)

10	devsel_3	DEVSEL# must be deasserted after last transfer (PCI Spec. Appendix C, Rule 12 f)
11	trdy_0	DEVSEL# must be asserted with or prior to the edge at which the target enables its output (PCI Spec. Appendix C, Rule 34)
12	trdy_1	TRDY# must not go low the first clock after address phase in a read transaction (PCI Spec. Sect. 3.3.1. Read Transaction - see timing diagram)
13	trdy_2	once a target has asserted TRDY#, it cannot change DEVSEL#, TRDY# or STOP# until the current data phase completes (PCI Spec. Appendix C, Rule 12 d)
14	stop_0	DEVSEL# must be asserted with or prior to the edge at which the target enables its output (STOP#) (PCI Spec. Appendix C, Rule 34)
15	stop_1	once asserted, stop must remain asserted until FRAME# is deasserted whereupon STOP# must be deasserted (PCI Spec. Appendix C, Rule 12 c)
16	stop_2	once a target has asserted STOP# it cannot change DEVSEL#, STOP# or TRDY# until the current data phase completes (PCI Spec. Appendix C, Rule 12 d)
17	lock_0	LOCK# must be asserted the clock following the (first) address phase and kept to maintain control (PCI Spec. Appendix C, Rule 32 e)
18	lock_1	the first transaction of a locked access must be a read. (PCI Spec. Appendix C, Rule 32 d)
19	lock_2	LOCK# must be released if RETRY is signaled before data phase or whenever an access is terminated by target-abort or master-abort (PCI Spec. Appendix C, Rules 32 f and 32 g)
20	cache_0	after HITM, CLEAN must be signaled before STANDBY (PCI Spec. Sect. 3.9.2 Supported State Transitions)
21	cache_1	HITM must only be signaled after STANDBY (PCI Spec. Sect. 3.9.2 Supported State Transitions)
22	parity_0	PERR# may never be asserted two clocks after address phase or during a special cycle. During WRITE, PERR# may be asserted two clocks after IRDY#, during READ, PERR# may be asserted two clocks after TRDY#. (PCI Spec. Sect. 3.8.2. Error Reporting)
23	parity_1	address parity error (PCI Spec. Appendix C, Rule 37 b)
24	parity_2	a parity error has occurred, but it was not signaled

Pattern Terms

Pattern terms are logical combinations of PCI bus signals, bus states, protocol errors, master and target states and trigger input states which:

- provide a powerful trigger mechanism for the trace memory
- provide a sample qualifier for the trace memory
- provide a master conditional start pattern

Bus Observer

The bus observer provides information about the current bus state. This information is useful for triggering with one pattern only. The bus state is aligned to the PCI bus signals and is also sampled by the trace memory.

The bus observer statemachine provides the following outputs:

Bus observer outputs

Signals	State	Hex	Meaning
B_STATE[2:0]	unsync	0	After Reset the bus observer enters this state. As soon as an IDLE state occurs, the “idle” state is entered.
	idle	1	PCI IDLE
	dac1	2	First cycle of a dual address phase
	addr	3	Address phase
	dac2	4	Second cycle of a dual address phase
	decoding	5	Address phase has passed and no target has responded yet
	wait	6	Either master or target inserts waits.
	transfer	7,	Data transfer phase

Operator in order of decreasing precedencen

Operator	Operation	Applicable
!	negation	All types
==	Equality	All types
|	logical OR	List types only (“Available Pattern Terms” on page 144.)
&	logical AND	All types

Syntax Examples

The following should not be used:

```
!FRAME & AD32==b8xxx\h & CBE3_0==7\h
```

Rather, the following example could be used to trigger on a memory write address phase:

```
b_state==3\h & CBE3_0==7\h & AD32==b8xxx\h
```

xact_cmd can be used to trigger on a specific command. The following example

```
xact_cmd==7\h & (CBE3_0==7\h) & AD32==b8xxx\h
```

Note: if entering pattern from the C-API you need to enter a double backslash before the “h” character (for example, AD32==b8xxx\\h). This is C programming syntax.

Available Pattern Terms

Note: With pattern terms which contain “Don’t cares”, the don’t care bits must be rightmost in the entry field. For example *AD32==b8x00\h* is not allowed, but *AD32==b80xx\h* is OK.

Pattern Label	Signal	Type	Description
b_state	b_state[2:0]	List	state of bus tracking statemachine
t_act	t_act	10X	target active (this signal is asserted with devsel) ()
t_lock	t_lock	10X	target locked
m_act	m_act	10X	master involved ()
m_lock	m_lock	10X	master has lock

berr	berr	10X	summarized protocol error. This is generated by the analyzer's protocol checker. The protocol checker sets berr to 1 if any observed PCI rul is violated.
xact_cmd	xact_cmd[3:0]	List	command sampled during the last address phase
trigger3	trigger[3]	10X	external trigger input
trigger2	trigger[2]	10X	external trigger input
trigger1	trigger[1]	10X	external trigger input
trigger0	trigger[0]	10X	external trigger input
AD32	AD[31:0]	10X	PCI address & data
CBE3_0	C/BE[3:0]	10X	PCI command & byte enables
FRAME	FRAME#	10X	
TRDY	TRDY#	10X	
IRDY	IRDY#	10X	
DEVSEL	DEVSEL#	10X	
STOP	STOP#	10X	
IDSEL	IDSEL	10X	own IDSEL
PERR	PERR#	10X	
SERR	SERR#	10X	
REQ	REQ#	10X	REQ# of the own master
GNT	GNT#	10X	GNT# of own master
LOCK	LOCK#	10X	
SDONE	SDONE	10X	
SBO	SBO#	10X	
PAR	PAR	10X	
RST	RST#	10X	
INTA	INTA#	10X	
INTB	INTB#	10X	
INTC	INTC#	10X	
INTD	INTD#	10X	

Chapter 7 Programming Reference

This chapter contains an overview and a reference for all functions used by the C-API and the CLI

Objectives

C-API The C-API provides a programmatic interface to control the PCI Analyzer & Exerciser for automated test setups. It is possible to control several BEST devices using the C-API functions, regardless of which of the three possible controlling ports are used.

Command Line Interface (CLI) The CLI provides a means to use the C-API calls for simple interactive testing without using a C compiler. For typing convenience, each CLI command and associated parameters have an abbreviated form. The CLI is limited to one session handle.

Conventions

There are two types of parameters:

- Type (I) = Input parameter passed to the function
- Type (O) = Output parameter returned by the function.

For some parameters (e.g. the perr attribute) the parameter and value are optional. That means, if the parameter is typed to the command line without a value, the default for the value will be used. All functions return an error number.

Example:

```
MasterAttrPageProg   waits=12  last=0        // this will be the same as  perr=0
MasterAttrPageProg   waits=12  last=0  perr      // this sets perr=1
```

Naming of Constants Capital letters
Leading B_ to make them unique
Example: B_E_FILEOPEN

Naming of Types Lower case letters
leading 'b_' , trailing 'type'
Example: b_*err*type

Naming of Function Calls Each function name starts with a capital letter
Leading 'Best'
Action is expressed by the last word(s) in the call
Example: Best*Connect*

BestDevIdentifierGet ( )

Call	b_errtype BestDevIdentifierGet (b_int32 vendor_id, b_int32 device_id b_int32 number b_int32 *devid);
CLI equivalent	BestDevIdentifierGet vendor_id=*vendor_id* device_id=*device_id* snumber=*subsys_id*
CLI abbreviation	diget vendor=*vendor_id* dev=*device_id* n=*subsystem_id*
Description	This function returns the PCI device number for a PCI exerciser card when using PCI as the controlling interface port. The returned value of devid is then used as the port number in BestOpen() when using B_PORT_PCI as the communication port (communication over the PCI bus). If multiple cards are plugged into the system, the number stored in the subsystem id register in the configuration space can be used to distinguish between different exerciser cards. As this function is based upon Standard PCI BIOS calls, it can only be used in systems which support a Standard PCI BIOS. For other systems, the user is responsible for providing both the low level access functions to the configuration space of the exerciser card slot, and the device number. See *PCI Example on page 44* for an example.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298..
vendor_id	(I) vendor id of the exerciser card, default = 103C\h for HP
device_id	(I) device id for the exerciser card, default = 2925\h for HP E2925A exerciser
number	(I) index. This value is used to distinguish between different instances of the exerciser in one system (default = 0). The first card found has an index of 0, the second 1, etc..
***devid**	(O) pointer, where the device number is stored. This is the device number used to access the card's configuration space.

BestOpen ( )

Call

```
b_errtype BestOpen (b_handletype  * handle,
                    b_porttype    port,
                    b_int32       portnum );
```

CLI equivalent No equivalent. BestOpen() is called from the CLI during start-up. After that, port and portname are constant during the complete CLI session.

CLI abbreviation No equivalent.

Description This function initializes the internal structures and variables. It checks the connection to the BEST card by performing a write and read to/from an onboard register. After calling BestOpen() you must call BestConnect() with the returned handle to use the session. This function must be called before calling any other function. It returns a handle for a session to be used by subsequent C-API functions. The purpose of this function is to identify one unique exerciser card and to declare the control path for the session (e.g RS232, EPP, or PCI). It is possible to open multiple sessions for the same hardware (e.g. one RS232 and one PCI), however you cannot connect to more than one session at a time. You must perform a BestDisconnect() from one port before connecting to another. Multiple sessions for one hardware are not recommended.

For an example, *Opening and Closing the Connection to the Card on page 41*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (O) Handle to identify the session, comparable to a file handle. This handle is used in all subsequent C-API function calls.

port (I) This defines which communication mechanism is used to talk to the BEST card for the session.
This can be RS232, Parallel Centronics or the PCI Bus itself. See table below.

portnum (I) Identifies the specific port used to communicate. See "port and portnum" on page 151.

port and portnum

Port	Portnum	Description
B_PORT_RS232	B_PORT_COM1, B_PORT_COM2, B_PORT_COM3, B_PORT_COM4	Specifies one of four possible serial ports.
B_PORT_PARALLEL	B_PORT_LPT1, B_PORT_LPT2	Defines one of two parallel ports.
B_PORT_PCI_CONF	32 Bit	Device number of the BEST card, as used by the PCI BIOS and host bridge. This may be obtained using the function *BestDevIdentifierGet ( ) on page 149* in a system with PCI BIOS. If the system does not have a PCI BIOS then you must enter a specific system identifier to identify the BEST card.

BestRS232BaudRateSet ( )

Call

```
b_errtype BestRS232BaudRateSet (
                    b_handletype    handle,
                    b_int32         baudrate
            );
```

CLI equivalent BestRS232BaudRateSet baudrate=*baudrate*

CLI abbreviation brset baud=*baudrate*

Description This function can be used to change the baudrate from the default of 9600 baud, to a value between 9600 and 57600 baud.

For an example, *Opening and Closing the Connection to the Card on page 41*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session, comparable to a file handle

baudrate (I) valid baudrate entries are (CLI abbreviations in brackets):

- B_BD_9600 (9600)
- B_BD_19200 (19200)
- B_BD_38400 (38400)
- B_BD_57600 (57600)

BestConnect ( )

Call

```
b_errtype BestConnect (
                b_handletype    handle
                );
```

CLI equivalent BestConnect

CLI abbreviation con

Description This function is used to establish the link between the host and the BEST card, using the communication mechanism specified by the BestOpen() function.

The BEST card is controlled by this interface exclusively until BestDisconnect() is called.

As long as a connection has been established, connection requests from other ports are returned with a connect error.
For information on multiple sessions, "BestOpen ( )" on page 150.

If the function cannot connect successfully it returns B_E_CONNECT.

For an example, *Opening and Closing the Connection to the Card on page 41*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

BestDisconnect ( )

Call

```
b_errtype BestDisconnect (
                b_handletype    handle
                );
```

CLI equivalent BestDiconnect

CLI abbreviation discon

Description This function is used to close the link between the host and the BEST card. Disconnecting from one controlling interface allows a connection from another port.

If you are using multiple sessions (e.g. one RS232 and one PCI), then you must ensure that the BestsDisconnect() is complete before doing a BestConnect() to another port.
For information on multiple sessions, "BestOpen ( )" on page 150.

For an example, *Opening and Closing the Connection to the Card on page 41*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

BestClose ( )

Call

```
b_errtype BestClose(
                b_handletype    handle
        );
```

CLI equivalent No equivalent. Closing the CLI window executes BestClose() automatically.

Description This function closes the session and frees any allocated memory. If the serial port was used, then it also resets the baudrate to 9600.

For an example, *Opening and Closing the Connection to the Card on page 41*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle that identifies the session.

BestTestProtErrDetect ( )

Call

```
b_errtype  BestTestProtErrDetect (
           b_handletype   handle
           );
```

CLI equivalent BestTestProtErrDetect

CLI abbreviation testpedet

Description This function sets up the analyzer to behave as a PCI protocol error checker.
It performs the following onboard actions:

- clears the observer status register
- clears the OBS_RUN and TRC_RUN bits in the BEST status register.
- sets the analyzer trigger pattern to trigger on protocol errors
- starts the protocol observer and analyzer

The status is displayed by the Hex Display and can be read out using the *BestStatusRegGet ( ) on page 252*.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

BestTestResultDump ( )

Call	b_errtype BestTestResultDump (b_handletype handle, b_charptrtype filename);
CLI equivalent	BestTestResultDump
CLI abbreviation	testrdump file=filename
Description	This function saves the analyzer and observer status, including compare errors, in one file (<filename>.rpt), and the trace memory content in another file (<filename>.wfm). The trace memory file content can be post processed using the graphical user interface.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298..
handle	(I) handle to identify the session
***filename**	(I) char pointer to the path or filename.

BestTestPropSet ( )

Call	b_errtype BestTestPropSet (b_handletype handle, b_testproptype testprop, b_int32 value);
CLI equivalent	BestTestPropSet testprop=*testprop* value=*value*
CLI abbreviation	testprpset prop=*testprop* val=*value*
Description	This function sets a test property, used by a test function. See *Using the built-in test functions. on page 30* for a CLI example.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298..
handle	(I) handle to identify the session
testprop	(I) specifies the property to be set, see table below
value	(I) value the property is set to, see table below

b_testproptype

Properties/ (CLI abbreviation)	**Values/ (CLI abbreviations)**	**Description**
B_TST_BANDWIDTH (bw)	0 .. 100 (default 50)	Percentage of bus bandwidth requested.
B_TST_BLKLENGTH (blklen)	1/ .. 64k , default 8192.	Specifies the number of bytes for one block to be used as basis for the traffic. (dword aligned)

B_TST_DATAPATTERN (dpat)	B_DATAPATTERN_RANDOM (dprandom)/default	Sets a random data pattern for the tests.
	B_DATAPATTERN_FIX (dpfix)	Sets the content of the data buffer to be used by the test to 00000000\h.
	B_DATAPATTERN_TOGGLE (dptoggle)	Sets the content of the data buffer to alternate 00000000\h and FFFFFFFF\h.
B_TST_PROTOCOL (prot)	B_PROTOCOL_LITE (lite) / default	Sets the protocol variation to stress the system as little as possible.
	B_PROTOCOL_MEDIUM (medium)	Generate medium protocol stress.
	B_PROTOCOL_HARD (hard)	Generates maximum protocol stress.
B_TST_COMPARE (comp)	0 (default) / 1	1 forces the test function to do an onboard compare after a read.
B_TST_STARTADDR (start)	32 Bit default 00000000\h	sets the start address (dword aligned)
B_TST_NOFBYTES (nob)	32Bit default 4	Sets the blocksize for the test (dword aligned).
B_TST_SOURCEADDR (source)	32 Bit default 00000000\h	Source address (dword aligned)
B_TST_DESTINADDR (dest)	32 Bit default 00000000\h	Destination address (dword aligned)

BestTestPropDefaultSet ( )

Call	b_errtype BestTestPropSet (b_handletype handle,);
CLI equivalent	BestTestPropDefaultSet
CLI abbreviation	testprpdefset
Description	This function sets all the test properties to their default values. The default values are shown in section b testproptype on page 158. See *Using the built-in test functions. on page 30* for a CLI example.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298.
handle	(I) handle to identify the session

BestTestRun ( )

Call	b_errtype BestTestRun (b_handletype handle, b_int32 testcmd);
CLI equivalent	BestTestRun testcmd=*testcmd*
CLI abbreviation	testrun cmd=*testcmd*
Description	Starts the test function, which is specified by the test command (testcmd). The test runs once or loops infinitely, depending on the B_MGEN_REPEATMODE property. If the B_TST_COMPARE compare property is set, the test stops execution as soon as a data compare error occurs. See *Using the built-in test functions. on page 30* for a CLI example.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298..
handle	(I) handle to identify the session
testcmd	(I) The test command, See "testcmd" on page 162.

testcmd

Values/ (CLI abbreviations)	Description	Test properties utilized by this command
B_TSTCMD_TRAFFICMAKE (trafficmake)	Performs writes to its own target, thus consuming bus bandwidth.	B_TST_BANDWIDTH B_TST_BLKLENGTH B_TST_DATAPATTERN B_TST_PROTOCOL
B_TSTCMD_WRITEREAD (writeread)	Performs writes and reads to a system memory resource, specified by the start address	B_TST_BANDWIDTH B_TST_BLKLENGTH B_TST_DATAPATTERN B_TST_PROTOCOL B_TST_COMPARE B_TST_STARTADDR B_TST_NOFBYTES
B_TSTCMD_BLOCKMOVE (blockmove)	Moves a block of data from one system memory address to another by intermediately copying it to the internal data buffer.	B_TST_BANDWIDTH B_TST_BLKLENGTH B_TST_DATAPATTERN B_TST_PROTOCOL B_TST_COMPARE B_TST_SOURCEADDR B_TST_DESTINADDR B_TST_NOFBYTES
B_TSTCMD_READ (read)	Performs reads from a system memory resource.	B_TST_BANDWIDTH B_TST_BLKLENGTH B_TST_PROTOCOL B_TST_STARTADDR B_TST_NOFBYTES

BestMasterBlockPageInit ( )

Call

```
b_errtype BestMasterBlockPageInit (
                    b_handletype    handle,
                    b_int32         page_num
            );
```

CLI equivalent BestMasterBlockPageInit page_num=*page_num*

CLI abbreviation mbpginit [page=*page_num*]

Description This function initializes a master block transaction page, and sets the current block pointer to the beginning of the page. This function must be called once before a page can be programmed. The HP E2925A exerciser provides 128 block pages with 16 block entries per page. Pages are automatically concatenated when programmed accross page boundaries.

NOTE: This function will fail if it is called while a transaction is running.

See also "C-API Master Programming Model" on page 116

For an example, *Creating Master Transactions on page 46*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

page_num (I) The page number initialized. The block memory consists of 128 pages, therefore the page_num value has to be between 0 and 0xF.
Page 0 is used by the higher level C-API functions, and cannot be overwritten by the user.

BestMasterBlockPropDefaultSet ( )

Call b_errtype BestMasterBlockPropDefaultSet (
b_handletype handle,
);

CLI equivalent BestMasterBlockPropDefaultSet

CLI abbreviation mbprpdefset

Description This function is used to set the block preparation register to the default values.
The block preparation register is written to the current block by the BestMasterBlockProg() command.
See also “C-API Master Programming Model” on page 116

For an example, *Creating Master Transactions on page 46*

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle that identifies the session

Property	Value	Description
B_BLK_BUSADDR	0	PCI Bus address for the transfer
B_BLK_BUS_CMD	BUS_CMD_MEM_READ	PCI bus command
B_BLK_INTADDR	0	The BEST internal address for the transfer
B_BLK_BYTEN	0	The BYTEN value for the transfer
B_BLK_NOFDWORDS	1	The number of transfers
B_BLK_ATTRPAGE	0	The attribute page specifying protocol behavior

BestMasterBlockPropSet ( )

Call

```
b_errtype BestMasterBlockPropSet  (
                    b_handletype      handle,
                    b_blkproptype     blk_prop,
                    b_int32           value
          );
```

CLI equivalent BestMasterBlockPropSet blk_prop=*blk_prop* value=*value*

CLI abbreviation mbprpset prop=*blk_prop* val=*value*

Description This function is used to set an individual master block property (e.g. bus addresses) in the block preparation register. After all master block properties for a block transfer are set in the block preparation register, the complete block can be programmed into page memory using the BestMasterBlockProg() function.

The block transactions defined in the block preparation register may be run using the BestMasterBlockRun() function. Blocks programmed into page memory may be run using the BestMasterBlockPageRun() function.

Once programmed, the master block properties are stored on the BEST board. That means the property remains unchanged until it is reprogrammed.

For an example, *Creating Master Transactions on page 46*

See also “C-API Master Programming Model” on page 116

Return Value Error number or 0 if no error occurred, “Return Values” on page 298..

handle (I) handle that identifies the session

blk_prop (I) enumerated type, specifying the master block property to be set.
See “b_blkproptype” on page 166.

value (I) value to which the attribute property is set.

b_blkproptype

Properties/ (CLI abbreviation)	**Values / (CLI abbreviation)**	**Description**
B_BLK_BUSADDR (bad)	32Bit	specifies the physical PCI bus address, used as the starting address for the block transfer.
B_BLK_BUSCMD (cmd)		specifies the PCI bus command for the block transfer. (C/BE#[3:0] for address phase)
	B_CMD_INT_ACK (int_ack)	0x0
	B_CMD_SPECIAL (special)	0x1
	B_CMD_IO_READ (io_read)	0x2
	B_CMD_IO_WRITE (io_write)	0x3
	B_CMD_RESERVED_4 (reserved_4)	0x4
	B_CMD_RESERVED_5 (reserved_5)	0x5
	B_CMD_MEM_READ (mem_read)	0x6 (Default)
	B_CMD_MEM_WRITE (mem_write)	0x7
	B_CMD_RESERVED_8 (reserved_8)	0x8
	B_CMD_RESERVED_9 (reserved_9)	0x9
	B_CMD_CONFIG_READ (config_read)	0xA
	B_CMD_CONFIG_WRITE (config_write)	0xB
	B_CMD_MEM_READMULTIPLE, (mem_readmultiple)	0xC
	B_CMD_MEM_READLINE (mem_readline)	0xE
	B_CMD_MEM_WRITEINVALIDATE (mem_writeinvalidate)	0xF ***NOTE:*** **By using in combination with B_M_LAST illegal burst lengths could be generated.**
B_BLK_BYTEN (ben)	0x0 .. 0xF	PCI byte enables

B_BLK_INTADDR (iad)	0x0 .. 0x1FFFC	Dword-aligned byte address offset at the onboard data memory. See “Decoders and Internal Data Memory Model” on page 122.
B_BLK_NOFDWORDS (nod)	15Bit (default = 1)	Number of dwords to be transferred by the block transfer. Note: Which bytes are transferred is dependent on the byten enables.
B_BLK_ATTRPAGE (apage)	0(default) .. 255	Selects a master attribute page to define a PCI protocol behavior for the block.
B_BLK_COMPFLAG (cflag)	0(default)/1	A value of 1 means, that after execution of the block transfer, the data starting at B_BLK_INTADDR will be compared with the data starting at B_BLK_COMPOFFS. The result of the comparison is stored in the BestStatusRegister. The block transfer must be executed with the *BestMasterBlockPageRun ( )* function
B_BLK_COMPOFFS (coffs)	0(default) .. 0x1FFFC	Dword-aligned compare offset, see above.

BestMasterAllBlock1xProg ( )

Call

```
b_errtype BestMasterAllBlock1xProg  (
                    b_handletype    handle,
                    b_int32         busaddr,
                    b_int32         buscmd,
                    b_int32         byten,
                    b_int32         intaddr,
                    b_int32         nofdwords,
                    b_int32         attrpage,
                    b_int32         compflag,
                    b_int32         compoffset
                    );
```

CLI equivalent

BestMasterAllBlock1xProg busaddr=*busaddr* buscmd=*buscmd* byten=*byten* nofdwords=*nofdwords* intaddr=*intaddr* attrpage=*attrpage* compflag=*compflag* compoffset=*compoffset*

CLI abbreviation

mab1xprog bad=*busaddr* cmd=*buscmd* ben=*byten* iad=*intaddr* apage=*attrpage* cflag=*compflag* coffs=*compoffset* nod=nofdwords

Description

This function is used program a complete set of block properties for the HP E2925A exerciser to the current block page. This function is a convenience function and does not add functionality to the C-API.

This function calls BestMasterBlockPropDefaultSet(), then sets all properties by calling BestMasterBlockPropSet() once per property, and then calls BestMasterBlockProg().

NOTE: This function will fail if it is called while a transaction is running.

See also “C-API Master Programming Model” on page 116

Return Value

Error number or 0 if no error occurred, “Return Values” on page 298..

handle

(I) handle that identifies the session

Parameters The parameters are identical to the block property values describes in section BestMasterBlockPropSet ( ) on page 165

BestMasterBlockProg ( )

Call

```
b_errtype BestMasterBlockProg  (
                    b_handletype      handle,
                    );
```

CLI equivalent BestMasterBlockProg

CLI abbreviation mbprog

Description Calling this function programs the block attributes for one block of PCI transactions contained in the block preparation register to the current block page.
The block preparation register is programmed with all attributes using the BestMasterBlockPropSet() function prior to calling this function.

After programming the block, the block pointer is incremented by 1, so that it points to the next block in the current page. Successive calls of BestMasterBlockProg() can be used to program a linear sequence of block transfers.

For an example, *Creating Master Transactions on page 46*

NOTE: This function will fail if it is called while a transaction is running.

See also “C-API Master Programming Model” on page 116

handle (I) handle that identifies the session.

Return Value Error number or 0 if no error occurred, “Return Values” on page 298..

BestMasterBlockRun ( )

Call

```
b_errtype BestMasterBlockRun (
                b_handletype        handle,
                );
```

CLI equivalent BestMasterBlockRun

CLI abbreviation mbrun

Description This function starts a PCI block transfer defined by the block preparation register.

The function uses the block properties set previously using the BestMasterBlockPropSet() function.

The master starts the transfers on the bus corresponding to the run condition properties set previously using the BestMasterGenPropSet() function.

The function returns as soon the block transfer has started. To ensure that a block run has completed before disconnecting poll the staus of the master using *BestStatusRegGet ( )* until the master is inactive.

If a master abort occurs a compare error is generated, and if the compare flag is set execution stops.

If you want to do a block data comparison you must use *BestMasterBlockPageRun ( )*

NOTE: This function will fail if it is called while a transaction is running.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

BestMasterBlockPageRun ( )

Call

```
b_errtype BestMasterBlockPageRun (
                    b_handletype    handle,
                    b_int32         page_num
                    );
```

CLI equivalent BestMasterBlockPageRun page_num=*page*_num

CLI abbreviation mbpgrun page=*page_num*

Description This function runs the block page specified by page_num.

The master runs each block within the page in sequence. Each is run according to the run condition properties programmed previously using the BestMasterGenPropSet(). function.

The master starts the transfers on the bus corresponding to the run condition properties set previously using the BestMasterGenPropSet() function.

The function returns as soon the block transfer has started. To ensure that a page run has completed before disconnecting, poll the staus of the master using *BestStatusRegGet ( )* until the master is inactive.

If a master abort occurs a compare error is generated, and if the compare flag is set execution stops.

For an example, *Creating Master Transactions on page 46*

NOTE: This function will fail if it is called while a transaction is running.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

page_num (I) block page that is executed. All blocks defined in the page are executed in sequence.

BestMasterStop ( )

Call	b_errtype BestMasterStop (b_handletype handle,);
CLI equivalent	BestMasterStop
CLI abbreviation	mstop
Description	This function stops the current action of the master. The master stops the current transaction immediately after completing the current data transfer. For an example, *Creating Master Transactions on page 46*
Return Value	Error number or 0 if no error occurred, “Return Values” on page 298..
handle	(I) handle to identify the session

BestMasterAttrPageInit ( )

Call

```
b_errtype BestMasterAttrPageInit (
                    b_handletype    handle,
                    b_int32         page_num
            );
```

CLI equivalent BestMasterAttrPageInit page_num=*page_num*

CLI abbreviation mapginit page=*page_num*

Description This function initializes a master attribute memory page and sets the programming pointer to the beginning of the specified page.

This function must be called once before an attribute page can be programmed.

Note:Running a block page does not move the programming pointer.

Future exerciser hardware will have a maximum of 256 lines or phases each for master and target attributes, with mechanisms for repeating/ looping phases to maximize memory usage. If you wish to maintain compatibility with future exerciser hardware you should not use more than 256 protocol attribute phases/lines.

For an example, *Creating Master Transactions on page 46*

NOTE: This function will fail if it is called while a transaction is running.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

page_num (I) The number of the memory page to initialize. The attribute memory has 256 page entry points. Each page can contain up to 32 phases and can be concatenated.
Page zero is the default page, and cannot be overwritten by the user.
Valid entries are therefore 1 to 255.

BestMasterAttrPtrSet ( )

Call

```
b_errtype BestMasterAttrPtrSet(
                b_handletype    handle,
                b_int32         page_num,
                b_int32         offset
        );
```

CLI equivalent BestMasterAttrPtrSet page_num=*page_num* offset=*offset*

CLI abbreviation maptrset page=*page_num* offs=*offset*

Description This function is used to move the master attribute programming pointer to any offset relative to the start of the page.

Use this function to set the pointer to any data phase within attribute memory that you want to program.

Although attribute memory is organized into 256 pages of 32 lines (phases) each, pages may be programmed up to any offset within the limits of the attribute memory size. However, performing a BestMasterAttrPageInit() will initialize the specified page even if it has already been programmed. It is up to the programmer to take care of the attribute memory structure.

Other functions which move the attribute programming pointer are:
BestMasterAttrPageInit ( ) on page 174, moves it to the start of a page
BestMasterAttrPhaseProg ( ) on page 183, increments it to the next phase
BestMasterAllAttr1xProg ( ) on page 181, increments it to the next phase

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle handle that identifies the session

page_num (I) number of the attribute page to which the pointer should be set.
range 1 to 255

offset (I) offset relative to the start of the page

BestMasterAttrPropDefaultSet ( )

Call

```
b_errtype BestMasterAttrPropDefaultSet (
                b_handletype    handle,
                );
```

CLI equivalent BestMasterAttrPropDefaultSet

CLI abbreviation maprpdefset

Description This function is used to set the master attribute properties stored in the attribute preparation register to their default values.

The default values are then written to the line pointed to by the attribute programming pointer with the BestMasterAttrPhaseProg() function. The properties and their default values are described in section BestMasterAttrPropSet ( ) on page 177

For an example, *Creating Master Transactions on page 46*

Return Value Error number or 0 if no error occurred, “Return Values” on page 298..

handle (I) handle that identifies the session

BestMasterAttrPropSet ( )

Call

```
b_errtype BestMasterAttrPropSet  (
                    b_handletype        handle,
                    b_mattrproptype     mattrprop,
                    b_int32             value
                    );
```

CLI equivalent BestMasterAttrPropSet mattrprop=*mattrprop* value=*value*

CLI abbreviation maprpset prop=*mattrprop* val=*value*

Description This function is used to set an individual master attribute address or data phase property (e.g. number of waits) in the attribute preparation register. After all attribute properties for a dataphase are set, the complete phase attributes can be programmed into page memory using the BestMasterAttrPhaseProg() function. The properties stored in the attribute preparation register are used only for programming attribute page memory. They are not used as attributes when MasterBlockRun() is used. Once programmed, the master attribute properties are stored on the BEST board. That means the property remains unchanged until it is reprogrammed.
For an example, *Creating Master Transactions on page 46*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle that identifies the session

mattrprop (I) specifyies the master attribute property to be set. See "b_mattrproptype" on page 177.

value (I) value to which the attribute property should be set.

b_mattrproptype

Properties/ (CLI abbreviation)	Values	Description
B_M_DOLOOP (loop)	0(default)/1	1 sets the loop bit property, which forces the attribute structure to restart at the beginning of the page

B_M_WAITS (w)	0(default) .. 31	number of waits
B_M_LAST (last)	0(default)/1	1 defines the last data phase of a burst
B_M_DPERR (dperr)	0(default)/1	1 sets PERR# two clocks after the corresponding data phase, command register bit 6 must also be set.
B_M_DSERR (dserr)	0(default)/1	Assert SERR# two clocks after the corresponding data phase, command register bit 8 must also be set.
B_M_APERR (aperr)	0(default)/1	In an address phase, if *B_BOARD_PERREN* (page 276) and *B_BOARD_SERREN* (page 276) are set, aperr = 1 signals an address parity error using SERR# 2 cycles after this address phase. This is an address phase attribute.
B_M_DWRPAR (dwp)	0(default)/1	1 inverts the parity bit for the data phase
B_M_AWRPAR (awp)	0 (default)/1	1 inverts the parity bit in the corresponding address phase.
B_M_RELREQ (rreq)	B_DRELREQ_ON (on, 1)	forces the master to release REQ# in the corresponding address phase. This is an address phase attribute.
	B_DRELREQ_OFF (off, 0)/ default	the master will keep REQ# asserted as long as the intended action needs.
B_M_STEPMODE (stepmode)	B_STEPMODE_STABLE (stable, 0)	normal address phase
	B_STEPMODE_TOGGLE (toggle, 1)	the master performs 4 address steps in the address phase, while switching the address from non-inverted to inverted state.
B_M_WAITMODE (waitmode)	B_WAITMODE_STABLE (stable, 0)	normal data phase. Waits are determined by the waits parameter
	B_WAITMODE_TOGGLE (toggle, 1)	the master performs 4 data steps in the data phase, while switching the data value from non-inverted to the inverted state. Setting B_M_WAITMODE to this value sets B_M_WAITS to 4.

B_M_LOCK (lock)	B_LOCK_LOCK (lock, 1)	forces the master to try an exclusive access.
	B_LOCK_HIDELOCK (hidelock, 3)	the master performs an address phase, without releasing LOCK#, simulating an access to the locked target from another master.
	B_LOCK_UNCHANGE (unchange, 0)	keep the current LOCK# state unchanged
	B_LOCK_UNLOCK (unlock, 2) default	releases the LOCK# at the end of this transaction.

BestMasterAttrPropGet ( )

Call

```
b_errtype BestMasterAttrPropGet  (
                    b_handletype      handle,
                    b_mattrproptype   mattrprop,
                    b_int32           *value
                    );
```

CLI equivalent BestMasterAttrPropGet mattrprop=*mattrprop*
The CLI returns the property value to the CLI window.

CLI abbreviation maprpget prop=*mattrprop*

Description This function reads back a master attribute property from the attribute preparation register into the memory location specified by *value.

Return Value Error number or 0 if no error occurred, “Return Values” on page 298..

handle (I) handle that identifies the session

mattr (I) specifies the master attribute property to be read back.
See also “b_mattrproptype” on page 177

***value** (O) pointer to the attribute property value.

BestMasterAllAttr1xProg ( )

Call

```
b_errtype BestMasterAllAttr1xProg (
                    b_handletype    handle,
                    b_int32         doloop
                    b_int32         waits,
                    b_int32         last,
                    b_int32         dperr,
                    b_int32         dserr,
                    b_int32         aperr,
                    b_int32         dwrpar,
                    b_int32         awrpar,
                    b_int32         drelreq,
                    b_int32         stepmode,
                    b_int32         waitmode,
                    b_int32         lock
                    );
```

CLI equivalent Best MasterAllAttr1xProg doloop=*doloop* waits=*waits* last=*last* dperr=*dperr* dserr=*dserr* aperr=*aperr* dwrpar=*dwrpar* awrpar=*awrpar* drelreq=*drelreq* stepmode=*stepmode* waitmode=*waitmode* lock=*lock*

CLI abbreviation maa1xprog loop=*doloop* w=*waits* last=*last* dperr=*dperr* dserr=*dserr* aperr=*aperr* dwp=*dwrpar* awp=*awrpar* drreq=*drelreq* sm=*stepmode* wm=*waitmode* lock=*lock*

Description This function is used to program one complete line of master attribute properties to the current attribute page. This function is a convenience function and does not add functionality to the C-API. It calls the BestMasterAttrPropDefaultSet() function, then sets all properties by calling *BestMasterAttrPropSet ( ) on page 177* once per property, and then calls BestMasterAttrPhaseProg() to program the line (phase). See also “b_mattrproptype” on page 177

NOTE: This function will fail if it is called while a transaction is running.

Return Value Error number or 0 if no error occurred, “Return Values” on page 298..

handle (I) handle that identifies the session

BestMasterAttrPhaseProg ( )

Call

```
b_errtype BestMasterAttrPhaseProg  (
                    b_handletype      handle,
                    );
```

CLI equivalent BestMasterAttrPhaseProg

CLI abbreviation maphprog

Description This function programs the attributes for one PCI data phase to the memory location defined by the current attribute programming pointer. It uses the master attribute properties that have been set in the attribute preparation register.

After programming one phase, the attribute memory programming pointer is incremented to the next phase. Setting up the parameters for a complete transaction can be done by a sequence of BestMasterPhaseProg Calls.

The default attribute value for the loop bit is 0, which means that if you have less attribute phases than master block data phases you must remember to set the last phase loop bit attribute to 1. However, when the attribute memory page is initialized all phases are set to the default attributes, with the exception of the loop bit which is set. This means that if you forget to set the loop bit in the last phase the attribute phases will loop but with one more phase than was originally programmed.

For an example, *Creating Master Transactions on page 46*

NOTE: This function will fail if it is called while a transaction is running.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

BestMasterAttrPhaseRead ( )

Call

```
b_errtype BestMasterAttrPhaseRead  (
                    b_handletype      handle,
                    );
```

CLI equivalent BestMasterAttrPhaseRead

CLI abbreviation maphread

Description Reads the attributes from the current master attribute memory page location into the attribute preparation register.

NOTE: This function will fail if it is called while a transaction is running.

Return Value Error number or 0 if no error occurred, “Return Values” on page 298..

BestMasterGenPropSet ( )

Call

```
b_errtype BestMasterGenPropSet (
                b_handletype            handle,
                b_mastergenproptype     mastergenprop,
                b_int32                 value
                );
```

CLI equivalent BestMasterGenPropSet mastergenprop=*mastergenprop* value=*value*

CLI abbreviations mgprpset prop=*mastergenprop* val=*value*

Description This function sets a generic master run property. Once set, generic properties don't change during consecutive block runs.
Generic run properties, set the run mode (immediate or delayed) and master latency timer.
For more information on Latency Timer, "Master Latency Timer" on page 120..
For more information on Master Conditional Start, "Master Conditional Start" on page 121..

For an example, *Creating Master Transactions on page 46*

NOTE: This function will fail if it is called while a transaction is running.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session.

mastergenprop (I) the property to be set, see "b_mastergenproptype" on page 186.

value (I) value the property is set to.

b_mastergenproptype

Properties/ (CLI abbreviation)	Values/ (CLI abbreviations)	Description
B_MGEN_RUNMODE (runmode)	B_RUNMODE_IMMEDIATE (immediate, 0) / default	Sets the master to start without waiting on the master start condition.
	B_RUNMODE_WONDELAY (wondelay, 1)	Sets the master to wait until the master trigger pattern occurs, and the delay counter expires.
B_MGEN_REPEATMODE (repmode)	B_REPEATMODE_SINGLE (single, 1) / default	Programs the master to perform the Run one time.
	B_REPEATMODE_INFINITE (infinite, 0)	Programs the master to repeat Run until *BestMasterStop ( ) on page 173* is called.
B_MGEN_DELAYCTR (delayctr)	16 Bit, default = 0	Number of PCI clocks, between the occurrence of the conditional start pattern and the start of the master.
B_MGEN_LATMODE (latmode)	B_LATMODE_OFF (off)/default	This causes the master to either ignore the latency timer or to adhere to it by observing the proper PCI behaviour as outlined in the spec.
	B_LATMODE_ON	Switches the latency timer on.
B_MGEN_LATCTR (latctr)	0 to 255 clocks	Sets the latency timer value.
B_MGEN_MWIENABLE (mwien)	0 / 1	Sets memory write and invalidate enable bit in command register of configuration space.
B_MGEN_MASTERENABLE (men)	0 / 1	Sets memory master enable bit in command register of configuration space.
B_MGEN_ATTRMODE (attrmode)		This property defines when the master attribute pointer is reset to the beginning of the selected attribute page
	B_ATTRMODE_BLOCK (block, 0)/default	The master attribute pointer is set to the beginning of the selected attribute page each time a block is started
	B_ATTRMODE_PAGE (page, 2)	The master attribute pointer is set to the beginning of the selected attribute page when a block page is started
	B_ATTRMODE_SEQUENTIAL (sequential, 1)	The master attribute pointer is not changed when a new a new block or block page is started.

BestMasterGenPropDefaultSet ( )

Call	b_errtype BestMasterGenPropDefaultSet (b_handletype handle,);
CLI equivalent	BestMasterGenPropDefaultSet
CLI abbreviations	mgprpdefset
Description	This function sets the master generic properties to their default values. See “b_mastergenproptype” on page 186. For an example, *Creating Master Transactions on page 46* ***NOTE:*** This function will fail if it is called while a transaction is running.
Return Value	Error number or 0 if no error occurred, “Return Values” on page 298..
handle	(I) handle to identify the session

BestMasterGenPropGet ( )

Call

```
b_errtype BestMasterGenPropGet (
                    b_handletype          handle,
                    b_mastergenproptype   mastergenprop,
                    b_int32               *value
                    );
```

CLI equivalent BestMasterGenPropGet mastergenprop=*mastergenprop*

CLI abbreviation mgprpget prop=*mastergenprop*

Description This function reads master generic properties.

NOTE: This function will fail if it is called while a transaction is running.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

mastergenprop (I) the property to get, see "b_mastergenproptype" on page 186.

*** value** (O) pointer to the returned property value

BestMasterCondStartPattSet ( )

Call

```
b_errtype BestMasterCondStartPattSet (
                b_handletype    handle,
                b_charptrtype   pattern
                );
```

Description This function sets the master conditional start pattern (10X). This function allows you to conditionally start the master based on a bus event.

For an example, *Creating Master Transactions on page 46*

NOTE: This function will fail if it is called while a transaction is running.

CLI equivalent BestMasterCondStartPattSet pattern="*pattern*"

CLI abbreviation mcspset patt=*" pattern"*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

pattern (I) The pattern string that defines the compare pattern. Please refer to the chapter "Pattern Terms" on page 143. for details on pattern syntax.

BestTargetGenPropSet ( )

Call

```
b_errtype BestTargetGenPropSet (
                    b_handletype        handle,
                    b_targetgenproptype targetgenprop,
                    b_int32             value
                    );
```

CLI equivalent BestTargetGenPropSet targetgenprop=*targetgenprop* value=*value*

CLI abbreviation tgprpset prop=*targetgenprop* val=*value*

Description This function sets a generic target run property. Once set, generic properties don't change during consecutive target accesses.

For an example, *Creating Target Transactions on page 54*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

targetgenprop (I) the property to be set, see "b_targetgenproptype" on page 191.

value (I) value the property is set to, see "b_targetgenproptype" on page 191.

b_targetgenproptype

Properties/ (CLI abbreviation)	Values/ (CLI abbreviations)	Description
B_TGEN_RUNMODE (runmode)	B_RUNMODE_ADDRRESTART (addrrestart, 0)	Sets the target attribute structure to restart from the beginning of the page with every address phase.
	B_RUNMODE_SEQUENTIAL (sequential, 1), default	Sets the target to loop the attribute page only at the end of the page, thus providing a kind of random attribute set with respect to the address phases.
B_TGEN_MEMSPACE (memspace)	0 (default) / 1	Sets the Memory Space bit in the Configuration Space Command Register (bit 1). It is normally the job of the system configuration routine to enable/disable this bit during system initialization. The memory space decoders are not enabled unless this bit is set.
B_TGEN_IOSPACE (iospace)	0 (default) / 1	Sets the IO Space bit in the Configuration Space Command Register (bit 0). It is normally the job of the system configuration routine to enable/disable this bit during system initialization. The IO space decoders are not enabled unless this bit is set.
B_TGEN_ROMENABLE (romenable)	0 (default) / 1	A value 1 enables decoding of the expansion ROM

BestTargetGenPropGet ( )

Call

```
b_errtype BestTargetGenPropGet (
                b_handletype            handle,
                b_targetgenproptype     targetgenprop,
                b_int32                 *value
                );
```

CLI equivalent BestTargetGenPropGet targetgenprop=*targetgenprop*

CLI abbreviation tgprpget prop=*targetgenprop*

Description This function reads master generic properties
See “b_targetgenproptype” on page 191.

For an example, *Creating Target Transactions on page 54*

Return Value Error number or 0 if no error occurred, “Return Values” on page 298..

handle (I) handle to identify the session

targetgenprop (I) the property to get.

***value** (O) pointer to the returned value

BestTargetGenPropDefaultSet ( )

Call	b_errtype BestTargetGenPropDefaultSet (b_handletype handle,);
CLI equivalent	BestTargetGenPropDefaultSet
CLI abbreviation	tgprpdefset
Description	This function sets target generic properties to their default values. See “b_targetgenproptype” on page 191. For an example, *Creating Target Transactions on page 54*
Return Value	Error number or 0 if no error occurred, “Return Values” on page 298..
handle	(I) handle to identify the session

BestTargetDecoderPropSet ( )

Call	b_errtype BestTargetDecoderPropSet (b_handletype handle, b_decproptype decprop, b_int32 value);
CLI equivalent	BestTargetDecoderPropSet decprop=*decprop* value=*value*
CLI abbreviation	tdprpset prop=*decprop* val=*value*
Description	This function is used to set the properties of a decoder. After all properties for a particular decoder are set they are programmed using the *BestTargetDecoderProg ( )* function. To enable a decoder set the size to a non zero value. The card will then respond to decoded master accesses according to the base address (B_DEC_BASEADDR). For more information "Target Programming" on page 122.. For an example, *Creating Target Transactions on page 54*
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298..
handle	(I) handle to identify the session.
decprop	(I) Defines the property to be set, see "b_decproptype" on page 195.
value	(I) Value written to the specified property, see "b_decproptype" on page 195.

b_decproptype

Properties/ (CLI abbreviation)	Affects decoder	Values / (CLI abbreviation)	Description
B_DEC_MODE (mode)	1 - 6	B_MODE_MEM (mem) 0	Sets the specified decoder to respond to memory commands, valid for decoders 1 and 2.
		B_MODE_IO (io) 1	Sets the specified decoder to respond to I/O commands, valid for decoders 2 and 3.
B_DEC_SIZE (size)	1 2 3 7 8	0 [disabled], 12 to 24 0 [disabled], 4 to 16 (I/O) 0 [disabled], 6 to 16 (mem) 0 [disabled], 5 0 [disabled], 18 0 [disabled], 1 [enabled]	value is the power of 2 exponent for decoders1-3 programming registers in I/O space. expansion ROM 256kb config space 256 bytes
B_DEC_BASEADDR (base)	1 - 7	32 Bit	PCI base address, the address must have the same granularity as the programmed size.
B_DEC_SPEED (speed)	1,2	B_DSP_MEDIUM (medium) 1	Sets decoding speed to medium, valid for decoders 1 and 2.
		B_DSP_SLOW (slow) 0	Sets decoding speed to slow, valid for decoders 1 and 2.
B_DEC_LOCATION (loc)	1 - 6	B_LOC_SPACE32 (space32 0	If "mode=mem", sets to 32 bit address space
		B_LOC_BELOW1MEG (below1meg) 2	If "mode=mem", sets to below 1 Mb address space
		B_LOC_SPACE64 (space64) 4	If "mode=mem", sets to 64 bit address space
B_DEC_PREFETCH (prefetch)	1 - 6	0 (default) / 1	If "mode=mem",sets "prefetch" bit

BestTargetDecoder1xProg ( )

Call

```
b_errtype BestTargetDecoder1xProg (
                    b_handletype    handle,
                    b_int32         decoder_num,
                    b_int32         mode,
                    b_int32         size,
                    b_int32         base,
                    b_int32         speed
                    b_int32         location
                    b_int32         prefetch
                    );
```

CLI equivalent

BestTargetDecoder1xProg decoder_num=*decoder_num* size=*size* mode=*mode* base=*base_addr* speed=*speed* location=*location* prefetch=*prefetch*

CLI abbreviation

td1xprog dec=*decoder_num* size=*size* mode=*mode* base=*base_addr* speed=*speed* loc=*location* prefetch=*prefetch*

Description

This function sets and programs all properties for the specified decoder in one function call. This is a convenience function, based on the property set function.
For a description of property types and possible values "b_decproptype" on page 195..

For an example, *Creating Target Transactions on page 54*

NOTE: This function will fail if it is called while a transaction is running.

Return Value

Error number or 0 if no error occurred, "Return Values" on page 298..

handle

(I) handle to identify the session

decoder_num

(I) the decoder to program (1 - 8).

BestTargetDecoderPropGet ( )

Call

```
b_errtype BestTargetDecoderPropGet (
                    b_handletype    handle,
                    b_decproptype   decprop,
                    b_int32         *value
                    );
```

CLI equivalent BestTargetDecoderPropGet decprop=*decprop*

CLI abbreviation tdprpget prop=*decprop*

Description This function is used to read back a decoder property from the decoder preparation register into a host memory variable.
For a description of property types and possible values "b_decproptype" on page 195..

NOTE: This function will fail if it is called while a transaction is running.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle handle to identify the session, comparable to a file handle

decprop (I) Defines the property to get, see "b_decproptype" on page 195.

***value** (O) pointer to returned property value.

BestTargetDecoderProg ( )

Call

```
b_errtype BestTargetDecoderProg (
                    b_handletype    handle,
                    b_int32         decoder_num,
                    );
```

CLI equivalent BestTargetDecoderProg decoder_num=*decoder_num*

CLI abbreviation tdprog dec=*decoder_num*

Description This function is used to program decoder properties set by the *BestTargetDecoderPropSet ( )* function. This function checks that the property values in the target decoder preparation register are consistent with the specified decoder.

For an example, *Creating Target Transactions on page 54*

NOTE: This function will fail if it is called while a transaction is running.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

decoder_num (I) the decoder to be programmed (1 - 8).

BestTargetDecoderRead ( )

Call	b_errtype BestTargetDecoderRead (b_handletype handle, b_int32 decoder_num,);
CLI equivalent	BestTargetDecoderRead decoder_num=*decoder_num*
CLI abbreviation	tdread dec=*decoder_num*
Description	This function is used to read back a set of decoder properties into the decoder property preparation register. For a description of property types and possible values "b_decproptype" on page 195.. For an example, *Creating Target Transactions on page 54*
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298..
handle	handle to identify the session
decoder_num	(I) the decoder to be read (1 - 8).

BestTargetAttrPageInit ( )

Call

```
b_errtype BestTargetAttrPageInit (
                    b_handletype    handle,
                    b_int32         page_num
                    );
```

CLI equivalent BestTargetAttrPageInit page_num=*page_num*

CLI abbreviation tapginit page=*page_num*

Description This function initializes a target attribute memory page and sets the programming pointer to the beginning of the specified page.

This function must be called once before an attribute page can be programmed.

Page 0 is the target default page and cannot be programmed by the user.

Future exerciser hardware will have a maximum of 256 lines or phases each for master and target attributes, with mechanisms for repeating/ looping phases to maximize memory usage. If you wish to maintain compatibility with future exerciser hardware you should not use more than 256 protocol attribute phases/lines.

For an example, *Creating Target Transactions on page 54*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session.

page_num (I) The number of the memory page to initialize. The attribute memory has 256 page entry points. Each page can contain up to 32 target phases and can be concatenated.
Page zero is the default page, and cannot be overwritten by the user.
Valid entries are therefore 1 to 255.

BestTargetAttrPtrSet ( )

Call

```
b_errtype BestTargetAttrPtrSet(
                b_handletype    handle,
                b_int32         page_num,
                b_int32         offset
                );
```

CLI equivalent BestTargetAttrPtrSet page_num=*page_num* offset=*offset*

CLI abbreviation taptrset page=*page_num* offs=*offset*

Description This function is used to move the target attribute programming pointer to any offset relative to the start of the page. Use this function to set the pointer to any data phase within attribute memory that you want to program.

Although attribute memory is organized into 256 pages of 32 lines (phases) each, pages may be programmed up to any offset within the limits of the attribute memory size. However, performing a BestTargetAttrPageInit() will initialize the specified page (32 lines) even if it has already been programmed. It is up to the programmer to take care of the attribute memory structure.

Other functions which move the target attribute programming pointer are:

BestTargetDecoderProg ( ) on page 198, moves it to the start of a page
BestTargetAttrPhaseProg ( ) on page 207, increments it to the next phase
BestTargetAllAttr1xProg ( ) on page 206, increments it to the next phase

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle that identifies the session

page_num (I) number of the attribute page to which the pointer should be set.
range 1 to 255

offset (I) offset relative to the start of the page

BestTargetAttrPropDefaultSet ( )

Call	b_errtype BestTargetAttrPropDefaultSet (b_handletype handle,);
CLI equivalent	BestTargetAttrPropDefaultSet
CLI abbreviation	taprpdefset
Description	This function is used to set the target attribute properties stored in the attribute preparation register to their default values. The default values are then written to the line pointed to by the attribute programming pointer with the BestTargetAttrPhaseProg() function. The properties and their default values are described in section BestTargetAttrPropDefaultSet ( ) on page 202 For an example, *Creating Target Transactions on page 54*
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298..
handle	(I) handle that identifies the session

BestTargetAttrPropSet ( )

Call

```
b_errtype BestTargetAttrPropSet  (
                    b_handletype      handle,
                    b_tattrproptype   tattrprop,
                    b_int32           value
                    );
```

CLI equivalent BestTargetAttrPropSet tattrprop=*tattrprop* value=*value*

CLI abbreviation taprpset prop=*tattrprop* val=*value*

Description This function is used to set the target attribute properties on the exerciser board.
the target attribute properties are stored on the board in hardware.That means that, after being set, the property remains unchanged until it will be reprogrammed.
note: this function does not activate the target.

This function is used to set an individual target attribute data phase property (e.g. number of waits) in the attribute preparation register. After all attribute properties for a dataphase are set, the complete phase attributes can be programmed into page memory using the BestTargetAttrPhaseProg() function.

The properties stored in the attribute preparation register are used only for programming attribute page memory.

Once programmed, the target attribute properties are stored on the BEST board.
That means the property remains unchanged until it is reprogrammed.

For an example, *Creating Target Transactions on page 54*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle that identifies the session

tattrprop (I) specifies the target attribute property to be set, see "b_tattrproptype" on page 204.

value (I) value to which the attribute property should be set.

b_tattrproptype

Properties / (CLI abbreviations)	Value / (CLI abbreviations)	Description
B_T_DOLOOP (loop)	0 / 1, default is 0	1 sets the loop bit, which forces the target attribute structure to restart at the beginning of the attribute page
B_T_WAITS (w)	0 to 31, default= 0	number of waits, for reads the minimum initial waits is 2.
B_T_TERM (term)	B_TERM_NOTERM (noterm, 0)	default is no termination
	B_TERM_RETRY (retry, 1)	forces a retry in the corresponding data phase
	B_TERM_DISCONNECT (discon, 2)	forces a disconnect in the corresponding data phase
	B_TERM_ABORT (abort, 3)	forces a target abort in the corresponding data phase
B_T_DPERR (dperr)	0 / 1, default is 0	asserts PERR# for the corresponding data phase, bit 6 in the command register must also be set.
B_T_DSERR (dserr)	0 / 1, default is 0	asserts SERR# in the corresponding data phase.
B_T_APERR (aperr)	0 / 1, default is 0	asserts SERR# if the phase is an address phase
B_T_WRPAR (wp)	0 / 1, default is 0	inverts the parity bit in the corresponding data phase

BestTargetAttrPropGet ( )

Call

```
b_errtype BestTargetAttrPropGet  (
                    b_handletype      handle,
                    b_tattrproptype   tattrprop,
                    b_int32           *value
                    );
```

CLI equivalent BestTargetAttrPropGet tattrprop=*tattrprop*
The CLI returns the property value to the CLI window.

CLI abbreviation taprpget prop=*tattrprop*

Description This function reads back a target attribute property from the attribute preparation register into the memory location specified by *value.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle that identifies the session.

tattr (I) specifies the target attribute property to be read back.
See also "b_tattrproptype" on page 204

***value** (O) pointer to the attribute property value.

BestTargetAllAttr1xProg ( )

Call

```
b_errtype BestTargetAllAttr1xProg (
                    b_handletype    handle,
                    b_int32         doloop,
                    b_int32         waits,
                    b_int32         term,
                    b_int32         dperr,
                    b_int32         dserr,
                    b_int32         aperr,
                    b_int32         wrpar
                    );
```

CLI equivalent

BestTargetAllAttr1xProg
doloop=*doloop* waits=*waits* term=*term*
dperr=*dperr* dserr=*dserr* aperr=*aperr*
wrpar=*wrpar*

CLI abbreviation

taa1xprog
loop=*doloop* w=*waits* term=*term*
dperr=*dperr* dserr=*dserr* aperr=*aperr*
wp=*wrpar*

Description

This function is used to program one complete line of target attribute properties to the current attribute page.

This function is a convenience function and does not add functionality to the C-API.

It calls the BestTargetAttrPropDefaultSet() function, then sets all properties by calling BestTargetAttrPropSet() once per property, and then calls BestTargetAttrPhaseProg() to program the line (phase).

See also “b_tattrproptype” on page 204

Return Value

Error number or 0 if no error occurred, “Return Values” on page 298..

handle

(I) handle that identifies the session

BestTargetAttrPhaseProg ( )

Call	b_errtype BestTargetAttrPhaseProg (b_handletype handle,);
CLI equivalent	BestTargetAttrPhaseProg
CLI abbreviation	taphprog
Description	This function programs the attributes for one PCI target data phase to the memory location defined by the current attribute programming pointer. It uses the target attribute properties that have been set in the attribute preparation register. After programming one phase, the attribute memory programming pointer is incremented to the next phase. Setting up the parameters for a complete transaction can be done by a sequence of BestTargetPhaseProg() Calls. The default attribute value for the loop bit is 0, which means that if you have less attribute phases than target access phases you must remember to set the last phase loop bit attribute to 1. However, when the attribute memory page is initialized all phases are set to the default attributes, with the exception of the loop bit which is set. This means that if you forget to set the loop bit in the last phase the attribute phases will loop but with one more phase than was originally programmed. For an example, see “Creating Target Transactions” on page 54.
Return Value	Error number or 0 if no error occurred, “Return Values” on page 298..
handle	(I) handle that identifies the session

BestTargetAttrPhaseRead ( )

Call	b_errtype BestTargetAttrPhaseRead (b_handletype handle,);
CLI equivalent	BestTargetAttrPhaseRead
CLI abbreviation	taphread
Description	Reads the attributes from the current target attribute memory page location into the attribute preparation register. This function does not increment the programming pointer.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298..

BestTargetAttrPageSelect ( )

Call	b_errtype BestTargetAttrPageSelect (b_handletype handle, b_int32 page_num);
CLI equivalent	BestTargetAttrPageSelect page_num=*page_num*
CLI abbreviation	tapgsel page=*page_num*
Description	This function is used to select the target attribute page. This enables the target to respond when accessed from a master. The target attribute page 0 is programmed with the default behavior. This means turning on a decoder *BestTargetDecoderPropSet ( ) on page 194* activates the target without the need to program any attribute pages. For an example, see "Creating Target Transactions" on page 54.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298..
handle	(I) handle to identify the session.
page_num	(I) target attribute page to use, valid entries are 0 to 255 Page 0 is the default page and cannot be overwritten. Each page consists of 32 phase entries Pages greater than 32 phases can be created simply by continuing to program into the next page. However, if a page is initialized all phases in that page are overwritten with the default values.

BestObsMaskSet ( )

Call

```
b_errtype BestObsMaskSet (
                    b_handletype    handle,
                    b_obsruletype   obsrule,
                    b_int32         value
                    );
```

CLI equivalent BestObsMaskSet obsrule=*obsrule* value=*value*

CLI abbreviation omset rule=*obsrule* val=*value*

Description This function is used to mask out individual protocol errors. When a protocol error occurs, and it is not masked, the Observer Status Register (bit 2) is set, and the appropriate bit in the Accumulated error register is set.
For a definition of each error, see “Protocol Observer” on page 141.
See *Programming the Protocol Observer on page 64* for an example.

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

obsrule (I) the protocol rule to be masked, see “b_obsruletype” on page 211.

value (I) 0 = error not masked (default), 1 = error masked, see “b_obsruletype” on page 211.

b_obsruletype

Properties / (CLI abbreviations)	Properties / (CLI abbreviations)
B_R_FRAME_0 (frame_0)	B_R_TRDY_2 (trdy_2)
B_R_FRAME_1 (frame_1)	B_R_STOP_0 (stop_0)
B_R_IRDY_0 (irdy_0)	B_R_STOP_1 (stop_1)
B_R_IRDY_1 (irdy_1)	B_R_STOP_2 (stop_2)
B_R_IRDY_2 (irdy_2)	B_R_LOCK_0 (lock_0)
B_R_IRDY_3 (irdy_3)	B_R_LOCK_1 (lock_1)
B_R_IRDY_4 (irdy_4)	B_R_LOCK_2 (lock_2)
B_R_DEVSEL_0 (devsel_0)	B_R_CACHE_0 (cache_0)
B_R_DEVSEL_1 (devsel_1)	B_R_CACHE_1 (cache_1)
B_R_DEVSEL_2 (devsel_2)	B_R_PARITY_0 (par_0)
B_R_DEVSEL_3 (devsel_3)	B_R_PARITY_1 (par_1)
B_R_TRDY_0 (trdy_0	B_R_PARITY_2 (par_2)
B_R_TRDY_1 (trdy_1)	

BestObsMaskGet ( )

Call

```
b_errtype BestObsMaskGet (
                b_obsruletype   obsrule,
                b_int32         *value
                );
```

CLI equivalent BestObsMaskGet obsrule=*obsrule*

CLI abbreviation omget rule=*obsrule*

Description This function reads the mask bit of the specified rule error flag
See *Programming the Protocol Observer on page 64* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

obsrule (I) the protocol rule to be masked, "b_obsruletype" on page 211.

*** value** (O) pointer to the value returned by the function.

BestObsPropDefaultSet ( )

Call

```
b_errtype BestObsPropDefaultSet (
                b_handletype    handle
                );
```

Description This function sets the observer mask to zero, therefore enabling all protocol error checking. See *Programming the Protocol Observer on page 64* for an example.

CLI equivalent BestObsPropDefaultSet

CLI abbreviation oprpdefset

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

BestObsStatusGet ( )

Call

```
b_errtype BestObsStatusGet (
                    b_handletype    handle,
                    b_obsstatustype obsstatus,
                    b_int32         *value
                    );
```

Description

This function is used to readout the protocol observer status. Use this function to:

- Determine the first error that occurred during a run
- Determine all errors that occurred during a run
- Determine the observer status (running/stopped, errors/no errors)

To translate the value passed back into a meaningful text string (see section Accumulated Error Register and First Error Register (accuerr and firsterr) on page 215), use function *BestObsErrStringGet ( )*.
To reset the registers use function *BestObsStatusClear ( )*.
If the error mask bit is set, then the value is undefined.
See *Programming the Protocol Observer on page 64* for an example.

CLI equivalent

BestObsStatusGet obsstatus=*obsstatus*

CLI abbreviation

osget stat=*obsstatus*

Return Value

Error number or 0 if no error occurred, "Return Values" on page 298.

handle

(I) handle to identify the session

obsstatus

(I) defines which status register is read (first, accumulated or status), see "b_obsstatustype" on page 215.

***value**

(O) pointer to the 32Bit register value

b_obsstatustype

Properties/ (CLI abbreviations)	Description
B_OBS_FIRSTERR (firsterr)	The returned value indicates the first error that occurred after observer start
B_OBS_ACCUERR (accuerr)	The returned value indicates all protocol errors that occurred since the start of the observer.
B_OBS_OBSSTAT (obsstat)	The returned value is the observer status register.

Accumulated Error Register and First Error Register (accuerr and firsterr)

Error Type	Error Type	Error Type	Error Type
B_R_FRAME_0	B_R_DEVSEL_0	B_R_STOP_0	B_R_CACHE_1
B_R_FRAME_1	B_R_DEVSEL_1	B_R_STOP_1	B_R_PARITY_0
B_R_IRDY_0	B_R_DEVSEL_2	B_R_STOP_2	B_R_PARITY_1
B_R_IRDY_1	B_R_DEVSEL_3	B_R_LOCK_0	B_R_PARITY_2
B_R_IRDY_2	B_R_TRDY_0	B_R_LOCK_1	
B_R_IRDY_3	B_R_TRDY_1	B_R_LOCK_2	
B_R_IRDY_4	B_R_TRDY_2	B_R_CACHE_0	

Observer Status Register (obsstat)

Bit	Meaning	Bit	Meaning
[0]	1= observer is currently in runmode	[2]	1= protocol error detected
[1]	1= observer out of sync	[31:3]	not used, will return 0

BestObsErrStringGet ( )

Call

```
b_errtype BestObsErrStringGet (
                b_handletype    handle
                b_int32         bitposition,
                b_charptrtype   *errtext
                );
```

Description This function returns a text string relating to the specified error register bit position. For example, if bit position 2 in the error register is set, you must call this function with value "2". The function then passes back a pointer to the text: "IRDY must not be asserted on the same clock edge that FRAME# is asserted, but one or more clocks later".
See *Programming the Protocol Observer on page 64* for an example.

CLI equivalent BestObsErrStringGet bitposition=*bitposition*

CLI abbreviation oestrget pos=*bitposition*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

bitposition (I) the bit position set in one of the error registers.

***errstring** (O) a pointer to a pointer which points to the observer error rule type.

BestObsRuleGet ( )

Call

```
b_errtype BestObsRuleGet (
                b_int32         bitposition,
                b_obsruletype   *obsrule
                );
```

Description

This function returns a pointer to the error type relating to the specified error register bit position. For example, if bit position 2 in the error register is set, you must call this function with value "2". The function then passes back a pointer to the error type such as B_R_IRDY_0.

See “Accumulated Error Register and First Error Register (accuerr and firsterr)” on page 215.

For a description of rule types and their associated textual descriptions see “Protocol Observer” on page 141.
See *Programming the Protocol Observer on page 64* for an example.

CLI equivalent

No CLI equivalent, use *BestObsErrStringGet ( ) on page 216* instead.

Return Value

Error number or 0 if no error occurred, “Return Values” on page 298.

bitposition

(I) the bit position set in one of the error registers.

***obsrule**

(O) a pointer to the observer error rule type.

BestObsStatusClear ( )

Call

```
b_errtype BestObsStatusClear (
                b_handletype    handle
                );
```

Description This function clears the observer status register, and the first and accumulated error registers.
See “BestObsStatusGet ( )” on page 214.
See *Programming the Protocol Observer on page 64* for an example.

CLI equivalent BestObsStatusClear

CLI abbreviation osclear

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

BestObsRun ( )

Call

```
b_errtype BestObsRun (
                b_handletype    handle
                );
```

Description This function starts the protocol observer and sets bit 0 of the Observer Status Register. See *Programming the Protocol Observer on page 64* for an example.

CLI equivalent BestObsRun

CLI abbreviation orun

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

BestObsStop ( )

Call

```
b_errtype BestObsStop (
                b_handletype    handle
                );
```

Description This function stops the protocol observer and resets bit 0 of the Observer Status Register. The contents of the error registers remain unchanged.
See *Programming the Protocol Observer on page 64* for an example.

CLI equivalent BestObsStop

CLI abbreviation ostop

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

BestTracePropSet ( )

Call

```
b_errtype BestTracePropSet (
                b_handletype      handle,
                b_traceproptype   traceprop,
                b_int32           value
                );
```

Description This function sets a property for the analyzer trace memory. The trace memory can be triggered on detection of a bus pattern (normal mode) or when an expected event does not occur (heartbeat mode). See “Analyzer Overview” on page 140.
See *Triggering the Analyzer Trace Memory on page 60* for an example.

CLI equivalent BestTracePropSet traceprop=*traceprop* value=*value*

CLI abbreviation trcprpset prop=*traceprop* val=*value*

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

traceprop (I) specifies the property to be set, see “b_traceproptype” on page 221.

value (I) value the property is set to.

b_traceproptype

Property values/ (CLI abbreviations)	Values/ (CLI abbreviations)	Description
B_TRC_HEARTBEATMODE (hbmode)	B_HBMODE_ON (on, 1	Switches trigger generation to heartbeatmode
	B_HBMODE_OFF (off, 0) / default	Normal trigger mode

B_TRC_HEARTBEATVALUE (hbvalue)	16Bit	Heartbeat trigger value in PCI clocks, granularity 2^8

BestTracePattPropSet ( )

Call

```
b_errtype BestTracePattPropSet (
        b_handletype            handle,
        b_tracepattproptype     traceprop,
        b_charptrtype           pattern
        );
```

Description This function sets the compare pattern (10X) for the trace memory trigger and sample qualifier.
See *Triggering the Analyzer Trace Memory on page 60* for an example.

CLI equivalent BestTracePropStringSet traceprop=*traceprop* pattern=*"pattern"*

CLI abbreviation trcpprpset prop=*traceprop* patt=*"pattern"*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

traceprop (I) specifies the property to be set.

pattern (I) is a pattern string, defining the 10X pattern. Please refer to *Pattern Terms on page 143* to get a detailed description of the pattern term string syntax.

b_tracepattproptype

Property values/ (CLI abbreviations)	Values	Description
B_PT_TRIGGER (trig)	"pattern"	Trigger pattern, see "Pattern Terms" on page 143.
B_PT_SQ (sq)	"pattern"	Sample qualifier, see "Pattern Terms" on page 143.

BestTraceDataGet ( )

Call

```
b_errtype BestTraceDataGet (
                b_handletype    handle,
                b_int32         startline,
                b_int32         n_of_lines,
                b_int32         *data
                );
```

Description This function is used to load logic analyzer trace memory from the board. Before using BestTraceDataGet, *BestTraceStop ( )* or *BestAnalyzerStop ( )* must be called.
See *Triggering the Analyzer Trace Memory on page 60* for an example.

CLI equivalent BestTraceDataGet startline=*startline* n_of_lines=*n_of_lines*

CLI abbreviation trcdget start=*startline* nol=*n_of_lines*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

startline (I) specifies the start address of onboard logic analyzer trace memory

n_of_lines (I) specifies the quantity in lines of data to be uploaded. One line of analyzer data is defined by the *BestTraceBytePerLineGet ( )* and is dependent on the product hardware you are using.

***data** (O) pointer to array of 32Bit values, where the analyzer data will be stored. This array contains a maximum of 32k states which occupy 3 dwords each.

BestTraceBitPosGet ( )

Call

```
b_errtype BestTraceBitPosGet (
                b_handletype    handle,
                b_signaltype    signal,
                b_int32         *position,
                b_int32         *length
                );
```

Description This function is used to return the position and length of the specified signal. See also "BestTraceBytePerLineGet ( )" on page 227.
See *Triggering the Analyzer Trace Memory on page 60* for an example.

CLI equivalent BestTraceBitPosGet signal=*signal*

CLI abbreviation trcbtposget signal=*signal*

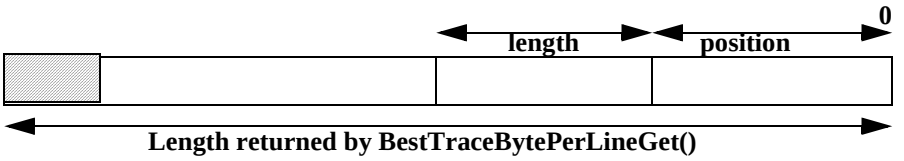
Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

signal (I) specifies the signal

***position** (O) the bit position, within an analyzer trace state, of the specified signal. This value is the offset (in bits) from the least significant bit to the first bit of the specified signal.

***length** (O) the length in bits of the signal data. This is the width of the signal.



b_signaltype

B_SIG_AD32	B_SIG_CBE3_0	B_SIG_FRAME	B_SIG_IRDY	B_SIG_TRDY
B_SIG_DEVSEL	B_SIG_STOP	B_SIG_IDSEL	B_SIG_PERR	B_SIG_SERR
B_SIG_REQ	B_SIG_GNT	B_SIG_LOCK	B_SIG_SDONE	B_SIG_SBO
B_SIG_PAR	B_SIG_RESET	B_SIG_BERR	B_SIG_INTA	B_SIG_INTB
B_SIG_INTC	B_SIG_INTD	B_SIG_trigger15	B_SIG_trigger14	B_SIG_trigger13
B_SIG_trigger12	B_SIG_trigger11	B_SIG_trigger10	B_SIG_trigger9	B_SIG_trigger8
B_SIG_trigger7	B_SIG_trigger6	B_SIG_trigger5	B_SIG_trigger4	B_SIG_trigger3
B_SIG_trigger2	B_SIG_trigger1	B_SIG_trigger0	B_SIG_b_state	B_SIG_m_act
B_SIG_t_act	B_SIG_m_lock	B_SIG_t_lock	B_SIG_samplequal	

B_SIG_b_state 3 bits defining the bus state, see “Bus Observer” on page 143.

B_SIG_m_act single bit which is 1 when the master is active.

B_SIG_t_act single bit which is 1 when the target is active.

B_SIG_m_lock single bit which is 1 when the master is performing a locked transaction.

B_SIG_t_lock single bit which is 1 when the target is locked by a bus master.

B_SIG_samplequal single bit which is 0 when the previous stored sample was not the previous PCI state. That is, there is a gap in captured samples.

BestTraceBytePerLineGet ( )

Call

```
b_errtype BestTraceBytePerLineGet (
                    b_handletype    handle,
                    b_int32         *bytes_per_line
                    );
```

Description This function is used to get the bytes per captured analyzer line (state). This value is dependent on the BEST hardware you are using.
See *Triggering the Analyzer Trace Memory on page 60* for an example.

CLI equivalent BestTraceBytePerLineGet

CLI abbreviation trcbtplget

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

bytesperline (O) returns the number bytes per line of trace data for the BEST hardware you are using

BestTraceStatusGet ( )

Call

```
b_errtype BestTraceStatusGet (
                    b_handletype       handle,
                    b_tracestatustype  tracestatus,
                    b_int32            *status
                    );
```

Description This function is used to read the following information from the analyzer trace memory:

- status register
- line number of the trigger event
- number of captured lines

See *Triggering the Analyzer Trace Memory on page 60* for an example.

CLI equivalent BestTraceStatusGet tracestatus=*tracestatus*

CLI abbreviation tsget stat=*tracestatus*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

tracestatus (I) specifies the status to be read, see "b_tracestatustype" on page 229.

***status** (O) pointer to the returned status information

b_tracestatustype

Properties/ (CLI abbreviation)	Description
B_TRC_STAT (stat)	Returns the content of the Trace Status Register. See Trace Status Register below:
B_TRC_TRIGPOINT (trig)	Returns the line number corresponding to the trigger event.. [a]
B_TRC_LINESCAPT (lines)	Returns the number of lines captured. [a]

a. Can only be read out after calling *BestAnalyzerStop ( )* or *BestTraceStop ( )*.

Trace Status Register

Bit	Description
[0]	1= trace memory stopped acquiring
[1]	1= trigger occurred
[2:31]	not used

BestTraceRun ( )

Call

```
b_errtype BestTraceRun (
                b_handletype    handle
                );
```

Description This function enables the trace memory. Data is acquired according to the trigger mode and storage qualifier properties.
See *Triggering the Analyzer Trace Memory on page 60* for an example.

CLI equivalent BestTraceRun

CLI abbreviation trcrun

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

BestTraceStop ( )

Call

```
b_errtype BestTraceStop (
                b_handletype    handle
                );
```

Description This function stops the current trace run. The current run status information or trace memory content are not affected.
See *Triggering the Analyzer Trace Memory on page 60* for an example.

CLI equivalent BestTraceStop

CLI abbreviation trcstop

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

BestAnalyzerRun ( )

Call

```
b_errtype BestAnalyzerRun (
                b_handletype    handle
                );
```

Description This function starts the PCI analyzer. This includes the protocol observer and the trace memory.

CLI equivalent BestAnalyzerRun

CLI abbreviation arun

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

BestAnalyzerStop ( )

Call

```
b_errtype BestAnalyzerStop (
                b_handletype    handle
                );
```

Description This function stops the PCI analyzer (protocol observer and trace memory), the current run status information and trace memory content are not affected.

CLI equivalent BestAnalyzerStop

CLI abbreviation astop

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

BestStaticPropSet ( )

Call

```
b_errtype BestStaticPropSet (
                    b_handletype      handle,
                    b_int32           pin_num,
                    b_staticproptype  staticprop,
                    b_int32           value
          );
```

CLI equivalent BestStaticPropSet pin_num=*pin_num* staticprop=*staticprop* value=*value*

CLI abbreviation sprpset pin=*pin_num* prop=*staticprop* val=*value*

Description This function configures each pin of the 8 Bit static IO port as either:

- Input only
- Open-drain
- Totem-pole output.

The default pin type is "Input only", therefore if another pin type is required this function must be called first. For more information on Static IO, see “Static I/O Port” on page 315.
See *Using the Static I/O port on page 74* for an example.

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

pin_num (I) pin to be configured. Valid entries are 0 to 7

staticprop (I) the specifiaction of the corresponding pin.
For valid entries, see “b_staticproptype” on page 235.

value (I) value of the specified property, see table below.

b_staticproptype

Properties/ (CLI abbreviation)	Values / (CLI abbreviations)	Description
B_STAT_PINMODE (pinmode)	B_PMD_INPONLY (inponly, 0) / default	Specifies the corresponding pin as input only
	B_PMD_TOTEMPOLE (totempole, 1)	Specifies the corresponding pin as totem-pole
	B_PMD_OPENDRAIN (opendrain, 2)	Specifies the corresponding pin as open-drain

BestStaticWrite ( )

Call

```
b_errtype BestStaticWrite (
                b_handletype    handle,
                b_int32         value
                );
```

CLI equivalent BestStaticWrite value=*value*

CLI abbreviation swrite val=*value*

Description This function sets the logical value of all Static IO signals in one function call. The lowest 8 bits of the value written correspond to the 8 pins.
See *Using the Static I/O port on page 74* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

value value to be written (no default)

BestStaticRead ( )

Call	b_errtype BestStaticRead (b_handletype handle, b_int32 *value);
CLI equivalent	BestStaticRead
CLI abbreviation	sread
Description	This function reads the Static IO port buffer and returns a byte data value. See *Using the Static I/O port on page 74* for an example.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298.
handle	(I) handle to identify the session
***value**	(O) pointer to the static IO port value.

BestStaticPinWrite ( )

Call

```
b_errtype BestStaticPinWrite (
                b_handletype    handle,
                b_int32         pin_num,
                b_int32         value
                );
```

CLI equivalent BestStaticPinWrite pin_num=*pin_num* value=*value*

CLI abbreviation spwrite pin=*pin_num* val=*value*

Description This function sets the logical value of one single Static IO pin.
All other pins remain the same.
See *Using the Static I/O port on page 74* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

pin_num (I) identifies the pin to program, valid entries are 0 to 7. There is no default.

value (I) value set, valid entries are 0, 1 or 2. When a value of 2 is used the current value at the pin is inverted for approximately 130ms. and then restored to its previous value.

BestCPUportPropSet ( )

Call

```
b_errtype BestCPUportPropSet (
                    b_handletype    handle,
                    b_cpuproptype   cpuprop,
                    b_int32         value
          );
```

CLI equivalent BestCPUportPropSet cpuprop=*cpuprop* value=*value*

CLI abbreviation cpuprpset prop=*cpuprop* val=*value*

Description This function configures the CPUport properties.
See *Using the CPU port on page 70* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

cpuprop (I) defines the property to set, see "b_cpuproptype" on page 240.

value (I) value set, see table below.

b_cpuproptype

Properties/ (CLI abbreviation)	Values/ (CLI abbreviation)	Description
B_CPU_MODE (mode)	B_CM_MASTER (master, 0)	Sets the CPUport mode to master.
	B_CM_DISABLED (disabled, 2) / default	Disables the CPUport and sets the outputs to high-impedance.
B_CPU_PROTOCOL (proto)	B_CP_INTEL (intel, 0) / default	Sets the protocol type to be Intel compatible.
B_CPU_RDYTYPE (rdy)	B_CR_EXTERNAL (external, 0)	Sets the RDY# signal generation to external.
	B_CR_AUTO (auto, 1) / default	Sets the RDY# signal generation to internal.

BestCPUportWrite ( )

Call

```
b_errtype BestCPUportWrite (
                b_handletype    handle,
                b_int32         device_num,
                b_int32         address,
                b_int32         data,
                b_sizetype      size
                );
```

CLI equivalent BestCPUportWrite device_num=*device_num* address=*address* data=*data* size=*size*

CLI abbreviation cpuwrite dev=*device_num* ad=*address* val=*data* size=*size*

Description This function writes data to the CPU port while asserting the specified address on the CPU ports address lines. The target of the access is specified by the device_num parameter. The CPU performs the write according to the currently enabled protocol type.
See “b_cpuproptype” on page 240.
See *Using the CPU port on page 70* for an example.

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

device_num (I) defines one of the 2 possible target devices for the access.
This has the effect of setting the corresponding Sel# lines, valid entries are 0 or 1

address (I) address at the address lines of the CPU port, valid entries 0 to 64k

data (I) value written

size (I) size of the data to be written. Possible sizes are:

- B_SIZE_BYTE - 8 bits (CLI abbreviation = byte)
- B_SIZE_WORD - 16 bits (CLI abbreviation = word)

BestCPUportRead ( )

Call

```
b_errtype BestCPUportRead (
                    b_handletype    handle,
                    b_int32         device_num,
                    b_int32         address,
                    b_int32         *data_ptr,
                    b_sizetype      size
                    );
```

CLI equivalent BestCPUportRead device_num=*device_num* address=*address* size=*size*

CLI abbreviation cpuread dev=*device_num* ad=*address* size=*size*

Description This function reads data from the CPU port while asserting the specified address on the CPU ports address lines. The target device for the access is selected by the device_num parameter. The CPU performs the read according to the currently enabled protocol type.
See “b_cpuproptype” on page 240.
See *Using the CPU port on page 70* for an example.

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

device_num (I) defines one of the 2 possible target devices for the access.
This has the effect of setting the corresponding Sel# lines, valid entries are 0 or 1.

address (I) address at the address lines of the CPU port, valid entries 0 to 64k

data_ptr (O) pointer to the returned data

size (I) size of the data to be read. Possible sizes are:

- B_SIZE_BYTE - 8 bits (CLI abbreviation = byte)
- B_SIZE_WORD - 16 bits (CLI abbreviation = word)

BestCPUportIntrStatusGet ( )

Call	b_errtype BestCPUportIntrStatusGet (b_handletype handle, b_int32 *intvalue_ptr);
CLI equivalent	BestCPUportIntrStatusGet
CLI abbreviation	cpuistatget
Description	This function reads the CPU port latched Interrupt state. See *Using the CPU port on page 70* for an example.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298.
handle	(I) handle to identify the session
intvalue_ptr	(I) pointer to the status of the CPU ports Interrupt Line. Returned value is 0 or 1.

BestCPUportIntrClear ( )

Call	b_errtype BestCPUportIntrClear (b_handletype handle,);
CLI equivalent	BestCPUportIntrClear
CLI abbreviation	cpuintclear
Description	This function clears the CPU port interrupt latch. See *Using the CPU port on page 70* for an example.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298.
handle	(I) handle to identify the session

BestCPUportRST ( )

Call

```
b_errtype BestCPUportRST (
                b_handletype    handle,
                b_int32         value,
                );
```

CLI equivalent BestCPUportRST value=*value*

CLI abbreviation cpurst val=*value*

Description This function sets the Reset signal to the rst_value. To pulse the CPU Port RST#, you must make successive calls to this function, the first call sets the value to 0, and the second call back to 1. This allows programming of the pulse duration using a timed loop between the calls. See *Using the CPU port on page 70* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

value (I) logical value of the Reset signal (0/1)

BestConfRegSet ( )

Call

```
b_errtype BestConfRegSet (
                b_handletype    handle,
                b_int32         offset,
                b_int32         value
                );
```

CLI equivalent BestConfRegSet offset=*offset* value=*value*

CLI abbreviation conrset offs=*offset* val=*value*

Description This function sets the specified configuration register to the specified value.
This enables writing to configuration space without having to use configuration type accesses.
See also "Configuration Space" on page 128
This value only becomes the power up default after calling *BestAllPropStore ( ) on page 265*

NOTE: This function will fail if it is called while a transaction is running

See *Programming the configuration space of the E2925A on page 84* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I)handle to identify the session

offset (I) address offset in the configuration space
Entries should be on DWORD address boundaries 00 - 3C\h

value (I) 32 Bit data value written to the register

BestConfRegGet ( )

Call

```
b_errtype BestConfRegGet (
                    b_handletype    handle,
                    b_int32         offset,
                    b_int32         *value
                    );
```

CLI equivalent BestConfRegGet offset=offset

CLI abbreviation conrget offs=offset

Description This function returns the specified configuration register value.
This enables reading of the configuration space header without having to use configuration type accesses. See also “Configuration Space” on page 128
See *Using the onboard expansion ROM on page 87* for an example.

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

offset (I) address offset into configuration space
Entries should be on DWORD address boundaries 00 - 3C\h

value (I) Pointer to the 32 Bit data value read

BestConfRegMaskSet ( )

Call

```
b_errtype BestConfRegMaskSet (
                    b_handletype    handle,
                    b_int32         offset,
                    b_int32         value
                    );
```

CLI equivalent BestConfRegMaskGet offset=*offset* value=*maskvalue*

CLI abbreviation conrmaskset offs=*offset* val=*maskvalue*

Description This function is used to define which registers in the configuration space header are read-only for configuration accesses.
See *Programming the configuration space of the E2925A on page 84* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle handle to identify the session

offset (I) address offset into the configuration space header
Entries should be on DWORD address boundaries 00 - 3C\h

maskvalue (O) 32 bit mask: 0 = RO bit , 1 = RW bit.

BestConfRegMaskGet ( )

Call

```
b_errtype BestConfRegMaskGet (
                    b_handletype    handle,
                    b_int32         offset,
                    b_int32         *value
                    );
```

CLI equivalent BestConfRegMaskSet offset=*offset*

CLI abbreviation conrmaskset offs=*offset*

Description This function is used to determine which registers in the configuration space header are read-only for configuration accesses.
This mask only becomes the power up default after calling *BestAllPropStore ( ) on page 265*

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle handle to identify the session

offset (I) address offset into the configuration space header
Entries should be on DWORD address boundaries 00 - 3C\h

maskvalue (O) 32 bit mask: 0 = RO bit , 1 = RW bit.

BestExpRomByteWrite ( )

Call

```
b_errtype BestExpRomByteWrite (
                b_handletype    handle,
                b_int32         offset,
                b_int32         value
                );
```

CLI equivalent BestExpRomByteWrite offset=*offset* value=*value*

CLI abbreviation erbytewrite offs=*offset* val=*value*

Description This function writes one byte to the specified offset in the expansion EEPROM. This allows the Expansion EEPROM to be used to test expansion ROM handling during POST.

NOTE: The upper 1Kbyte is used for internal purposes and should not be used.

See *Using the onboard expansion ROM on page 87* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

offset (I) byte address offset in the expansion ROM.
Valid entries are from 0000\h to FFFF\h (64kByte)

value (I) data byte written

BestExpRomByteRead ( )

Call

```
b_errtype BestExpRomByteRead (
                    b_handletype    handle,
                    b_int32          offset,
                    b_int32          *value
                    );
```

CLI equivalent BestExpRomByteRead offset=*offset*

CLI abbreviation erbyteread offs=*offset*

Description This function reads one byte from the specified address offset in the expansion EEPROM.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session, comparable to a file handle

offset (I) byte address offset in the expansion EEPROM.
valid entries are from 00000\h to FFFF\h (64kByte)

value (O) pointer to the returned data byte

BestStatusRegGet ( )

Call	b_errtype BestStatusRegGet (b_handletype handle, b_int32 *value_ptr);
CLI equivalent	BestStatusRegGet
CLI abbreviation	sregget
Description	This function reads the content of the BEST onboard status register to the value_ptr. This is a different register than PCI Configuration Space Header Status register. See *Handling Errors on page 41* for an example.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298.
handle	(I) handle to identify the session
value_ptr	(I) pointer to the returned value, see "BEST Status Register" on page 253.

BEST Status Register

Bit	Type	Default	Description	Values
[0]	RO	0	Master Run Bit	1= master active
[1]	RO	0	Target Active Bit	1= target active
[2]	RO	0	Observer Run Bit	1= observer running
[3]	RO	0	Trace Run Bit	1= trace memory in run mode
[4]	RO	0	Protocol Error	1= protocol error detected
[5]	RC	0	Data Compare Error	1= data compare error detected
[6]	RC	0	Functional Error	1= functional error during command
[7]	RC	0	Block aborted	1= at least one block has not been completely executed
[8]	RC	0	INTA asserted	1= asserted
[9]	RC	0	INTB asserted	1= asserted
[10]	RC	0	INTC asserted	1= asserted
[11]	RC	0	INTD asserted	1= asserted
[12]	RO	0	test run failed	1= test run has failed
[13:31]	RO	0	reserved	

BestStatusRegClear ( )

Call

```
b_errtype BestStatusRegClear (
                    b_handletype    handle,
                    b_int32          clearpattern
                    );
```

CLI equivalent BestStatusRegClear clearpattern=*clearpattern*

CLI abbreviation sregclear clear=*clearpattern*

Description A ’1’ in the clearpattern value clears the corresponding status bit in the BEST status register.
See also “BestStatusRegGet ( )” on page 252
See *Handling Errors on page 41* for an example.

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

clearpattern (I) a binary or hex pattern which defines the bits to be cleared in the BEST status register. A '1' clears the corresponding status bit. For example: *clear=11111111\b*, clears all bits.

BestInterruptGenerate ( )

Call

```
b_errtype BestInterruptGenerate (
                    b_handletype    handle,
                    b_int32         pci_int
                    );
```

CLI equivalent BestInterruptGenerate pci_int=*pci_int*

CLI abbreviation intgen int=*pci_int*

Description Sets the specified PCI interrupt pin. Because the values are constants, multiple interrupts can be set by ORing the int values (for example, *pci_int=B_INTA | B_INTC*). An interrupt is reset by clearing the corresponding status bit in the BEST status register ("BestStatusRegClear ( )" on page 254.).

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

int (I) Possible values are:

- B_INTA (inta)
- B_INTB (intb)
- B_INTC (intc)
- B_INTD (intd)

BestMailboxSendRegWrite ( )

Call

```
b_errtype BestMailboxSendRegWrite (
                    b_handletype    handle,
                    b_int32         value,
                    b_int32         *status
                    );
```

CLI equivalent BestMailboxSendRegWrite value=*value*

CLI abbreviation msregwrite val=*value*

Description This function writes the value to the onboard mailbox send register and passes back the status of the write. If this bit is set, then the send register still contains data that has not yet been read by the other participant. If this is the case then the data is not written to the send register. For more information on the Mailbox Registers, see "Mailbox Registers" on page 309.

NOTE: This function can only be used with an external control interface (serial or parallel interface).

See *Using the onboard expansion ROM on page 87* for an example demonstrating the mailbox functions.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

value (I) message byte

***status** (O) pointer to the status bit

BestMailboxReceiveRegRead ( )

Call	b_errtype BestMailboxReceiveRegRead (b_handletype handle, b_int32 * value, b_int32 * status);
CLI equivalent	BestMailboxReceiveRegRead
CLI abbreviation	mrregread
Description	This function reads the value of the receive mailbox register. Reading the mailbox receive register clears the receive register status bit. For more information on the Mailbox Registers, see "Mailbox Registers" on page 309. ***NOTE:*** This function can only be used with an external control interface (serial or parallel interface). See *Using the onboard expansion ROM on page 87* for an example demonstrating the mailbox functions.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298.
handle	(I) handle to identify the session
*value	(O) pointer to the received message data byte
*status	(O) pointer to the read status bit passed back by the function. If the status bit is set the data value returned is a previously unread message.

BestPCICfgMailboxSendRegWrite ( )

Call

```
b_errtype BestMailboxSendRegWrite (
                b_int32         devid,
                b_int32         value,
                b_int32         *status
                );
```

Description

This function writes the value to the onboard mailbox send register through the PCI config. space and passes back the status of the write. If this bit is set, then the send register still contains data that has not yet been read by the other participant. If this is the case then the data is not written to the send register.
For more information on the Mailbox Registers, see "Mailbox Registers" on page 309.

NOTE: This function can only be used with the PCI control interface..

See *Using the onboard expansion ROM on page 87* for an example demonstrating the mailbox functions.

Return Value

Error number or 0 if no error occurred, "Return Values" on page 298.

devid

(I) PCI device id of BEST

value

(I) message byte

***status**

(O) pointer to the status bit

BestPCICfgMailboxReceiveRegRead ( )

Call

```
b_errtype BestMailboxReceiveRegRead (
                b_int32         devid,
                b_int32         * value,
                b_int32         * status
                );
```

Description

This function reads the value of the receive mailbox register through the PCI config. space. Reading the mailbox receive register clears the receive register status bit.
For more information on the Mailbox Registers, see “Mailbox Registers” on page 309.

NOTE: This function can only be used with the PCI control interface..

See *Using the onboard expansion ROM on page 87* for an example demonstrating the mailbox functions.

Return Value

Error number or 0 if no error occurred, “Return Values” on page 298.

devid

(I) PCI device id of BEST

*value

(O) pointer to the received message data byte

*status

(O) pointer to the read status bit passed back by the function. If the status bit is set the data value returned is a previously unread message.

BestDisplayPropSet ( )

Call

```
b_errtype BestDisplayPropSet (
                    b_handletype    handle,
                    b_int32         value
                    );
```

CLI equivalent BestDisplayPropSet value=*value*

CLI abbreviation dprpset val=*value*

Description This function sets the mode of the Hex Display.
If in card mode, the display shows the error number of any detected protocol error.
If in user mode, the display can be written to by the BestDisplayWrite() function.
See *Using the Hex display on page 77* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

value (I) B_DISP_USER (CLI abbreviation - user)
B_DISP_CARD, default (CLI abbreviation - card)

BestDisplayWrite ( )

Call

```
b_errtype BestDisplayWrite (
                b_handletype    handle,
                b_int32         value
                );
```

CLI equivalent BestDisplayWrite value=*value*

CLI abbreviation dwrite val=*value*

Description This function sets the two digits of the Hex Display.
The display mode must be set to B_DISP_USER ("BestDisplayPropSet ( )" on page 260.) before this function can be used.
See *Using the Hex display on page 77* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

value (I) 8Bit value displayed

BestPowerUpPropSet ( )

Call

```
b_errtype BestPowerUpPropSet  (
                    b_handletype    handle,
                    b_puproptype    pu_prop,
                    b_int32         value
                    );
```

CLI equivalent BestPowerUpPropSet pu_prop=*pu_prop*value=*value*

CLI abbreviation puprpset prop=*pu_prop* val=*value*

Description This function is used to set the power up properties. These properties are stored into the onboard EEPROM by calling *BestAllPropStore ( )*. After power up, the board is automatically loaded with the values stored in the EEPROM. See also "Power-Up Behavior" on page 136 See *Using the onboard expansion ROM on page 87* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

pu_prop (I) specifies the power up property to be set, see table below

value (I) value to which the attribute is set.

b_puproptype

Properties/ (CLI abbreviation)	Values / (CLI abbreviation)	Description
B_PU_OBSRUNMODE (obsrunmode)	0 (default) /1	1 means that the observer will be set to run mode after power up.
B_PU_TRCRUNMODE (trcrunmode)	0 (default) /1	1 means that the trace memory will be set to run mode after power up.

B_PU_CONFRESTORE (confrestore)	0 (default) /1	1 means that the current values in the configuration space will be used when BestAllPropStore() is executed. 0 means that read/write bits are loaded from the factory default settings when BestAllPropStore() is executed.
B_PU_SSTRUNMODE (sstrunmode)	0 (default) /1	This property is used by the HP E2974A system stress test software. 1 means that an HP E2974A test will be run at powerup. 0 means that no subsytem test will be run at powerup. This property should not be directly set to 1, but it it may be set to 0 to disable a test which has already been set up using the HP E2974A GUI.

BestPowerUpPropGet ( )

Call

```
b_errtype BestPowerUpPropGet  (
                        b_handletype      handle,
                        b_puproptype      pu_prop,
                        b_int32           *value
                );
```

CLI equivalent BestPowerUpPropGet pu_prop=*pu_prop*value=*value*

CLI abbreviation puprpget prop=*pu_prop* val=*value*

Description This function is used to read back the current value of a specific power up property.
See “b_puproptype” on page 262.
See also “Power-Up Behavior” on page 136

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

pu_prop (I) specifies the power up property to be read

***value** (O) pointer to the returned property value.

BestAllPropStore ( )

Call	b_errtype BestBoardPropStore (b_handletype handle,);
CLI equivalent	BestAllPropStore
CLI abbreviation	aprpstore
Description	This function stores the complete set of properties to the user default part of the initialization property space in the onboard EEPROM. These properties then are used by the initialization function during power-up or by *BestAllPropLoad ( )*, to restore the board status. See also “Power-Up Behavior” on page 136 See *Using the onboard expansion ROM on page 87* for an example.
Return Value	Error number or 0 if no error occurred, “Return Values” on page 298.
handle	(I) handle to identify the session

BestAllPropLoad ( )

Call

```
b_errtype BestAllPropLoad  (
                    b_handletype      handle,
                    );
```

CLI equivalent BestAllPropLoad

CLI abbreviation aprpload

Description This function restores the complete set of user default properties from the initialization property space in the onboard EEPROMSee also “Power-Up Behavior” on page 136
The following user default properties are loaded:

- Master block property register (“b_blkproptype” on page 166.)
- Master attribute property register (“b_mattrproptype” on page 177.)
- Master generic properties (“b_mastergenproptype” on page 186.)
- Target generic properties (“b_targetgenproptype” on page 191.)
- Target decoder properties (“b_decproptype” on page 195.)
- Target attribute property register (“b_tattrproptype” on page 204.)
- Observer rule mask (“b_obsruletype” on page 211.)
- Analyzer trace trigger mode (“b_traceproptype” on page 221.)
- Analyzer trace triggers patterns (“b_tracepattproptype” on page 223.)
- Static IO pin types and outputs (“b_staticproptype” on page 235.)
- CPU port configuration (“b_cpuproptype” on page 240.)
- Power up properties (“b_puproptype” on page 262.)

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

BestAllPropDefaultLoad ( )

Call

```
b_errtype BestBoardPropDefaultLoad  (
                    b_handletype      handle,
                    );
```

CLI equivalent BestAllPropDefaultLoad

CLI abbreviation aprpdefload

Description This function restores the complete set of factory default properties from the initialization property space in the onboard CPU Flash ROM.

The following factory default properties are loaded:

- Master block property register (“b_blkproptype” on page 166.)
- Master attribute property register (“b_mattrproptype” on page 177.)
- Master generic properties (“b_mastergenproptype” on page 186.)
- Target generic properties (“b_targetgenproptype” on page 191.)
- Target decoder properties (“b_decproptype” on page 195.)
- Target attribute property register (“b_tattrproptype” on page 204.)
- Observer rule mask (“b_obsruletype” on page 211.)
- Analyzer trace trigger mode (“b_traceproptype” on page 221.)
- Analyzer trace triggers patterns (“b_tracepattproptype” on page 223.)
- Static IO pin types and outputs (“b_staticproptype” on page 235.)
- CPU port configuration (“b_cpuproptype” on page 240.)
- Power up properties (“b_puproptype” on page 262.)

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

BestDummyRegWrite ( )

Call

```
b_errtype BestDummyRegWrite (
                    b_handletype    handle,
                    b_int32         register_value
                    );
```

CLI equivalent BestBasicRegWrite register_value=*register_value*

CLI abbreviation bdrw val = *register_value*

Description This function is used to write a data value to a dummy register in the internal register space of the hardware. The purpose of this function is for users who need to write their own PCI driver (e.g. for a non-Intel platform) for communicating with the BEST card. It enables a simple method of testing the driver using a C-API function by making a simple write and read back test. This dummy register has no associated functionality.
As all C-API functions are based upon the same principle as these two dummy functions, they provide function an ideal mechanism for testing all functionality on a different platform.
See also *Porting the PCI interface driver to a UNIX platform on page 94* .

Return Value Error number or 0 if no error occurred, "Return Values" on page 298.

handle (I) handle to identify the session

register_value (I) data value

BestDummyRegRead ( )

Call	b_errtype BestDummyRegRead (b_handletype handle, b_int32 *register_value);
CLI equivalent	BestDummyRegRead
CLI abbreviation	bdrr
Description	This function reads the data value from the internal dummy register register. See “BestDummyRegWrite ( )” on page 268. See also *Porting the PCI interface driver to a UNIX platform on page 94* .
Return Value	Error number or 0 if no error occurred, “Return Values” on page 298.
handle	(I) handle to identify the session
***register_value**	(O) pointer to returned data value

BestErrorStringGet ( )

Call	b_charptrtype BestErrorStringGet (b_errtype error);
CLI equivalent	No CLI equivalent. The CLI display window provides this functionality.
CLI abbreviation	No CLI equivalent.
Description	This function returns a character pointer to an error string corresponding to the error number passed to it. See also *Handling Errors on page 41* .
Return Value	A pointer to a character string containing the error message.
handle	(I) handle to identify the session
errorm	(I) specifies the error returned by another C-API function, “Return Values” on page 298.

BestVersionGet ( )

Call	b_errtype BestVersionGet (b_handletype handle, b_versionproptype versionprop, b_charptrtype *string);
CLI equivalent	BestVersionGet versionprop=*versionprop*
CLI abbreviation	vget prop=*versionprop*
Description	This function is used to read the version/date information stored on the boards EEPROMs in order to check consistency between the firmware/HW and the C-API code.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298.
handle	(I) handle to identify the session
versionprop	(I) specifies the version property to be read back. see "b_versionproptype" on page 272.
***string**	(O) pointer to the version/date string. This string is statically allocated and is overwritten each time this function is called.

b_versionproptype

Properties/ (CLI abbreviation)	Description
B_VER_PRODUCT (product)	Returns the boards product number (e.g. E2925A)
B_VER_SERIAL (serial)	Returns the serial number of this specific BEST hardware
B_VER_BOARD (board)	Returns the PC board version (e.g. E2925C)
B_VER_SMDATE (smdate)	Returns the date code of the statemachine unit FPGA architecture file
B_VER_DPDATE (dpdate)	Returns the date code of the datapath unit FPGA architecture file
B_VER_LATTDATE (lattdate)	Returns the date code of the lattice component
B_VER_XILDATE (xildate)	Returns the date code of the Xilinx FPGA chain architecture file
B_VER_FIRMWARE (firmware)	Returns the date code of the onboard BIOS
B_VER_CORE (core)	Returns the date code of the onboard CORE-BIOS

BestSMReset ( )

Call	b_errtype BestSMReset (b_handletype handle);
CLI equivalent	BestSMReset
CLI abbreviation	smreset
Description	This function immediately resets the target, master and bus tracking statemachines. Because this function does not care about the any bus transactions that may be in progress, it should only be used in the case of an error.
Return Value	Error number or 0 if no error occurred, "Return Values" on page 298.
handle	(I) handle to identify the session

BestBoardReset ( )

Call	b_errtype BestBoardReset (b_handletype handle,);
CLI equivalent	BestBoardReset
CLI abbreviation	bdreset
Description	This function pulls the CPU reset for the onboard controller, thus forcing the board to reinitialize. This is equivalent to re-powering the board. See also “Power-Up Behavior” on page 136 See also “System Reset” on page 137
Return Value	Error number or 0 if no error occurred, “Return Values” on page 298.
handle	(I) handle to identify the session

BestBoardPropSet ( )

Call	b_errtype BestBoardPropSet (b_handletype handle, b_boardproptype boardprop, b_int32 value);
CLI equivalent	BestBoardPropSet boardprop=*boardprop* value=*value*
CLI abbreviation	bdprpset prop=*boardprop* val=*value*
Description	This function is currently used to set PCI RST# mode. See also “Power-Up Behavior” on page 136 See also “System Reset” on page 137 See also *Using the onboard expansion ROM on page 87* .
Return Value	Error number or 0 if no error occurred, “Return Values” on page 298.
handle	(I) handle to identify the session
boardprop	(I) specifies the board property to be set, see “b_boardproptype” on page 276.
value	(I) value, see table below.

b_boardproptype

Properties/ (CLI abbreviation)	Values / (CLI abbreviation)	Description
B_BOARD_RSTMODE (rstmode)	B_RSTMODE_RESETSM (resetsm, 1)	The PCI RST# signal will reset all statemachines, without affecting configuration space, decoders or other onboard properties.
	B_RSTMODE_RESETALL (resetall, 0) default	The PCI RST# signal will reinitialize the complete board. Equivalent to a hard reset or re-powering.
B_BOARD_PERREN (perren)	0 default / 1	A value 1 sets the configuration space command register bit 6, such that parity errors are reported normally.
B_BOARD_SERREN (serren)	0 default / 1	A value 1 sets the configuration space command register bit 8, such that system errors on SERR# are reported.
B_BOARD_ROMUSAGE (romusage)	B_ROMUSAGE_EXTERNAL (external, 0) default	The onboard EEPROM can be used as an expansion ROM.
	B_ROMUSAGE_INTERNAL (internal, 1)	The onboard EEPROM is used to store master and target attribute memory when BestAllPropStore is called. To set B_BOARD_ROMUSAGE to this property, the expansion ROM decoder must be disabled.

BestBoardPropGet ( )

Call

```
b_errtype BestBoardPropGet  (
                    b_handletype        handle,
                    b_boardproptype     boardprop,
                    b_int32             *value
            );
```

CLI equivalent BestBoardPropGet boardprop=*boardprop*

CLI abbreviation bdprpget prop=*boardprop*

Description This function is currently used to read the PCI RST# mode.
See also “Power-Up Behavior” on page 136
See also “System Reset” on page 137

Return Value Error number or 0 if no error occurred, “Return Values” on page 298.

handle (I) handle to identify the session

boardprop (I) specifies the board property to set, “b_boardproptype” on page 276.

***value** (O) pointer to the returned value

BestHostSysMemAccessPrepare ( )

Call

```
b_errtype BestHostSysMemAccessPrepare (
                    b_handletype    handle,
                    b_int32         buscmd,
                    b_int32         bufsize
                    );
```

CLI equivalent BestHostSysMemAccessPrepare buscmd=*cmd* bufsize=*bufsize*

CLI abbreviation hsmaprep cmd=*cmd* buf=*bufsize*

Description This function sets the master block properties int_addr and cmd and the BEST internal memory buffer size prior to calling *BestHostSysMemFill ( )* or *BestHostSysMemDump ( )* functions.
Using this function decreases download time when the fill or dump functions are called from within a program loop, because it decreases the register accesses performed by each fill and dump function call.

If using *BestHostSysMemFill ( )*, the cmd must be set to mem_write.
If using *BestHostSysMemDump ( )*, the cmd must be set to mem_read.
See *Using the Host access functions on page 67* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

cmd (I) specifies the bus command to be used.

bufsize (I) specifies the master data internal memory buffer size in dwords. This must be equal or larger than the buffersize specified in *BestHostSysMemFill ( )* or *BestHostSysMemDump ( )* functions.
valid range = 1 to 127.

BestHostSysMemFill ( )

Call

```
b_errtype BestHostSysMemFill (
                b_handletype    handle,
                b_int32         bus_addr,
                b_int32         num_of_bytes,
                b_int32         blocksize,
                int8 huge       *data_ptr
                );
```

CLI equivalent BestHostSysMemFill bus_addr=*bus_addr* num_of_bytes=*num_of_bytes* blocksize=*blocksize* data={*data list*}

CLI abbreviation hsmfill bad=*bus_addr* nob=*num_of_bytes* blk=*blocksize* data={*data list*}

Description This function moves a number of bytes of data from a buffer on the host system to the specified physical PCI bus address using 1 or more master block transfers.

It downloads the first blocksize chunk of data to BESTS internal memory through the controlling interface port. It then performs a master block transfer using memory write bursts to the specified system memory address. When the block transfer is complete the next blocksize chunk is transferred.

Each master block transfer can be one or more bursts depending on overall bus traffic and latency timer value.

NOTE: If a data transfer results in a master abort, this will be indicated in bit 7 (block aborted) of the status register. BestHostSysMemFill will *not* return an error. *BestStatusRegGet ( )* can be used to read the status register.

Function *BestHostSysMemAccessPrepare ( )* must be called prior to the first call of this function to set the cmd to mem_write and to set the BEST internal buffersize.
See *Using the Host access functions on page 67* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

data_ptr (I) C call only. This is a pointer to the source data in the host system memory. You cannot use a data pointer with the CLI, see "data" parameter below.

data CLI only. This parameter lists the data to be transferred (for example, data={1\h, 2\h, 3\h, 4\h, 5\h, 6\h, 7\h, 8\h} when using the CLI. This parameter replaces the data_ptr parameter used in the C call because the CLI cannot work with pointers. The data may also be imported from a file using a redirection operator (for example, data<"file path").
The input file consists of a sequence of hexadecimal byte values., e.g.:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

bus_addr (I) physical PCI bus address in PCI memory space where the data is written. Byte, word or dword addresses allowed.

n_of_bytes (I) specifies the total numbers of bytes to be transferred (maximum of 128kb in CLI, 4Gb with the C API).

blocksize (I) the size of the underlying master block transfers in bytes. This blocksize must be smaller or equal to the buffersize specified by the *BestHostSysMemAccessPrepare ( )* function.

BestHostSysMemDump ( )

Call	b_errtype BestHostSysMemDump (b_handletype handle, b_int32 bus_addr, b_int32 num_of_bytes, b_int32 blocksize, int8 huge *data_ptr);
CLI equivalent	BestHostSysMemDump bus_addr=*bus_addr* num_of_bytes=*num_of_bytes* blocksize=*blocksize* [data>"file path"]
CLI abbreviation	hsmdump bad=*bus_addr* nob=*num_of_bytes* blk=*blocksize [data>"file path"]*

Description This function moves a number of bytes of data from the specified physical PCI bus address, to a buffer on the host system using 1 or more master block transfers.

It performs a master block transfer of size "blocksize" using memory read bursts from the specified system memory address to BEST internal memory. It then uploads the blocksize chunk of data to the host memory address specified by data_ptr through the controlling interface port. When the upload is complete the next blocksize chunk is transferred.

Each master block transfer can be one or more bursts depending on overall bus traffic and latency timer value.

NOTE: If a data transfer results in a master abort, this will be indicated in the status register. BestHostSysMemDump will *not* return an error. *BestStatusRegGet ( )* can be used to read the status register.

Function *BestHostSysMemAccessPrepare ( )* must be called prior to the first call of this function to set the cmd to mem_read and to set the BEST internal buffersize.
The bus address can be a byte, word or dword address.
See *Using the Host access functions on page 67* for an example.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

data_ptr	( ) pointer to the destination in the host system memory. You cannot use a data pointer with the CLI, the data is either displayed in the window or directed to an output file.
data	CLI only. This parameter allows the data to be exported to a file using a redirection operator (for example, data>"file path").
bus_addr	( ) physical PCI bus address in PCI memory space from where the data is read. Byte, word or dword addresses allowed.
n_of_bytes	( ) specifies the total numbers of bytes to be transferred (maximum of 128kb in CLI, 4Gb with the C API).
blocksize	( ) the size of the underlying master block transfers in bytes. This blocksize must be smaller or equal to the buffersize specified by the *BestHostSysMemAccessPrepare ( )* function.

BestHostIntMemFill ( )

Call

```
b_errtype BestHostIntMemFill (
                b_handletype    handle,
                b_int32         int_addr,
                b_int32         num_of_bytes,
                b_int8 huge     *data_ptr
        );
```

CLI equivalent BestHostIntMemFill int_addr=*int_addr* num_of_bytes=*num_of_bytes* data={data list}

CLI abbreviation himfill iad=*int_addr* nob=*num_of_bytes* data={*data list*}

Description This function moves a block of data from a buffer on the host system to BEST internal memory. As the internal memory size is 128k bytes, this is the maximum number of bytes, which can be handled. This function can be used to initialize the data memory for further testing.
You should be aware that the internal memory is shared by the master and the PCI target memory and is physically the same memory.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

data_ptr (I) C call only. This is a pointer to the source data in the host system memory.
You cannot use a data pointer with the CLI, see "data" parameter below.

data CLI only. This parameter lists the data to be transferred (for example, data={1\h, 2\h, 3\h, 4\h, 5\h, 6\h, 7\h, 8\h} when using the CLI. This parameter replaces the data_ptr parameter used in the C call because the CLI cannot work with pointers. The data may also be imported from a file using a redirection operator (for example, data<"file path").
The input file consists of a sequence of byte values., e.g.:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

int_addr (I) dword-aligned byte address offset of BEST internal memory space.
Has to be between 0x0 and 0x1FFFC.

num_of_bytes (I) the total numbers of bytes (must be a multiple of 4) to be transferred. Has to be in the range from 1 to 0x20000.

BestHostIntMemDump ( )

Call

```
b_errtype BestHostIntMemDump (
                    b_handletype    handle,
                    b_int32         int_addr,
                    b_int32         num_of_bytes,
                    b_int8 huge     *data_ptr
          );
```

CLI equivalent BestHostIntMemDump int_addr=*int_addr* num_of_bytes=*num_of_bytes* [data>*"file path"*]

CLI abbreviation himdump iad=*int_addr* nob=*num_of_bytes* **[**data>"file path"]

Description This function moves a block of data from BEST internal memory to a buffer on the host. The number of bytes transferred is limited to 128k Bytes, which is the size of the BEST internal memory.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

data_ptr (I) pointer to the receive data buffer on the host

data CLI only. This parameter allows the data to be exported to a file using a redirection operator (for example, data>"file path").

int_addr (I) dword-aligned byte address offset of BEST internal memory space.
Valid entries are in the range 0x0 and 0x1FFFC.

num_of_bytes (I) specifies the total numbers of bytes (must be a multiple of 4) to be transferred.
Valid entries are in the range from 1 to 0x20000.

BestHostPCIRegSet ( )

Call

```
b_errtype BestHostPCIRegSet (
                    b_handletype        handle,
                    b_addrspacetype     addrspace,
                    b_int32             bus_addr,
                    b_int32             reg_value,
                    b_sizetype          wordsize
            );
```

CLI equivalent BestHostPCIRegSet addrspace=*addrspace*
bus_addr=*bus_addr*
reg_value=*reg_value* wordsize=*size*

CLI abbreviation hprgset space=*addr_space* bad=*bus_addr* val=*reg_value* size=*size*

Description This function writes the reg_value to the register specified by the addrspace and the bus address. The function performs a single cycle configuration write, memory write or I/O write depending upon the addrspace parameter.
The bus address is a byte address. The function automatically sets the right byte enables corresponding to the wordsize and the bus address. Accesses to BESTs own configuration register space should be done with the *BestConfRegSet ( )* and *BestConfRegGet ( )* functions.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

addrspace (I) specifies the address space for the register access, see "b_addrspacetype" on page 287.

bus_addr (I) physical PCI bus address.

reg_value (I) data to be written.

size (I) constant defining the register access as a byte, word or dword access. Possible values are:

- B_SIZE_BYTE
- B_SIZE_WORD
- B_SIZE_DWORD

b_addrspacetype

Properties/ (CLI abbreviations)	Description
B_ASP_CONFIG (config)	type 0 access to config space
B_ASP_CONFIG_TYPE1 (configtype1)	type 1 access to config space
B_ASP_IO (io)	access to io space
B_ASP_MEM (mem)	access to memory space

BestHostPCIRegGet ( )

Call

```
b_errtype BestHostPCIRegGet (
                    b_handletype      handle,
                    b_addrspacetype   addrspace,
                    b_int32           bus_addr,
                    b_int32           *regvalue_ptr,
                    b_sizetype        wordsize
                    );
```

CLI equivalent HostPCIRegGet addrspace=*addrspace* bus_addr=*bus_addr* wordsize=*size*
Note: the CLI command returns the reg_value to the CLI data window.

CLI abbreviation hprgget space=*addr_space* bad=*bus_addr* size=*size*

Description This function reads from the register specified by the address-space and the bus address. The function performs a single cycle configuration read, IO read or memory read depending on the address space parameter. The function sets the correct byte enables corresponding to the wordsize and the bus address boundary. If it is a word transfer, then the bus address must be at a word boundary. If it is a dword transfer, then the bus address must be at a dword boundary. Accesses to BESTs own configuration register space should be done with the *BestConfRegSet ( )* and *BestConfRegGet ( )* functions.

Return Value Error number or 0 if no error occurred, "Return Values" on page 298..

handle (I) handle to identify the session

addrspace (I) specifies the address space for the register access. Valid entries are shown in *b_addrspacetype on page 287*

bus_addr (I) physical PCI bus address.

regvalue_ptr (O) pointer to the returned register value

size (I) constant defining the register access as a byte, word or dword access. Possible values are:

B_SIZE_BYTE, B_SIZE_WORD and B_SIZE_DWORD

Chapter 8 Programming Quick Reference

This appendix provides a quick-reference for the C-API and CLI.

This appendix contains the following sections:

Overview of Programming Functions

Initialization and Connection Functions

BestDevIdentifierGet ( )	Returns a device id when using the PCI Bus as controlling interface, page 149
BestOpen ( )	Defines the controlling port and initializes internal structures and variables, page 150
BestRS232BaudRateSet ( )	Sets the RS232 baudrate if not using default of 9600, page 152
BestConnect ( )	Establishes an exclusive link between host and BEST hardware, page 153
BestDisconnect ( )	Disconnects the exclusive access link between host and BEST hardware, page 154
BestClose ( )	Closes the session and frees allocated memory, page 155

High Level Test Functions

BestTestProtErrDetect ( )	Initializes and starts the protocol observer to check for errors, page 156
BestTestResultDump ( )	Dumps the analyzer trace memory and observer status to file, page 157
BestTestPropSet ( )	Sets general purpose test properties (e.g. block transfer size), page 158
BestTestPropDefaultSet ( )	Sets the default properties, page 160
BestTestRun ( )	Starts a test function, page 161

Master Programming Functions

Block Transfer Functions	
BestMasterBlockPageInit ( )	Initializes a master block programming page, page 163 . This function will fail if it is called while a transaction is running
BestMasterBlockPropDefaultSet ( )	Sets the preparation block with the default values, page 164

BestMasterBlockPropSet ( )	Sets a preparation block property (e.g. bus command), page 165
BestMasterAllBlock1xProg ( )	Sets and programs the current block in one command, page 168 . This function will fail if it is called while a transaction is running
BestMasterBlockProg ( )	Programs the current block with the preparation block, page 170 This function will fail if it is called while a transaction is running
Run Functions	
BestMasterBlockRun ( )	Runs the block defined in the preparation block register, page 171 . This function will fail if it is called while a transaction is running.
BestMasterBlockPageRun ( )	Runs the specified block page, page 172 . This function will fail if it is called while a transaction is running
BestMasterStop ( )	Stops the currently running block, page 173
Protocol Behavior Functions	
BestMasterAttrPageInit ( )	Initializes a master attribute page, page 174 . This function will fail if it is called while a transaction is running
BestMasterAttrPtrSet ( )	Sets the attribute programming pointer, page 175
BestMasterAttrPropDefaultSet ( )	Sets the preparation register with the default values, page 176
BestMasterAttrPropSet ( )	Sets a property in the preparation register, page 177
BestMasterAttrPropGet ( )	Reads a propery value from the preparation register, page 180
BestMasterAllAttr1xProg ( )	Programs the current page block in one command, page 181 . This function will fail if it is called while a transaction is running
BestMasterAttrPhaseProg ( )	Programs the current block with the preparation register, page 183 . This function will fail if it is called while a transaction is running
BestMasterAttrPhaseRead ( )	Reads the current block into the preparation register, page 184 . This function will fail if it is called while a transaction is running

Generic Run Property Functions	
BestMasterGenPropSet ( )	Sets a generic run property, page 185 . This function will fail if it is called while a transaction is running
BestMasterGenPropDefaultSet ( )	Sets all generic run properties to their default values, page 187 . This function will fail if it is called while a transaction is running
BestMasterGenPropGet ( )	Reads a generic run property, page 188 . This function will fail if it is called while a transaction is running
BestMasterCondStartPattSet ( )	Sets a master conditional start pattern, page 189 . This function will fail if it is called while a transaction is running

Target Behavior and Decoder Programming Functions

Generic Properties	
BestTargetGenPropSet ( )	Sets a target run or generic decoder property, page 190
BestTargetGenPropGet ( )	Reads a target run or generic decoder property, page 192
BestTargetGenPropDefaultSet ( )	Sets all generic properties to their default values, page 193

Decoder Properties	
BestTargetDecoderPropSet ( )	Sets a single decoder property, enables a decoder, page 194
BestTargetDecoder1xProg ( )	Sets all decoder properties for 1 decoderr, page 196 . This function will fail if it is called while a transaction is running
BestTargetDecoderPropGet ( )	Reads a decoder property, page 197 . This function will fail if it is called while a transaction is running
BestTargetDecoderProg ( )	Programs a decoder, see page 198 . This function will fail if it is called while a transaction is running

BestTargetDecoderRead ( )	IREads the properties of a decoder into the , see page 199

Protocol Behavior	
BestTargetAttrPageInit ( )	Sets the attribute programming pointer, page 200
BestTargetAttrPtrSet ( )	Sets the attribute programming pointer, page 201
BestTargetAttrPropDefaultSet ( )	Sets the preparation register with the default values, page 202
BestTargetAttrPropSet ( )	Sets a property in the preparation register, page 203
BestTargetAttrPropGet ( )	Reads a propery value from the preparation register, page 205
BestTargetAllAttr1xProg ( )	Sets and programs the current page block in one command, page 206
BestTargetAttrPhaseProg ( )	Programs the current block with the preparation register, page 207
BestTargetAttrPhaseRead ( )	Reads the current block into the preparation register, page 208
BestTargetAttrPageSelect ( )	Selects the attribute page defining protocol behavior, page 209

PCI Protocol Observer Functions

These functions control the PCI Protocol Observer.

BestObsMaskSet ( )	Sets an individual error mask bit, page 210
BestObsMaskGet ( )	Reads an individual error mask bit, page 212
BestObsPropDefaultSet ( )	Sets all observer properties to their default values, page 213
BestObsStatusGet ( )	Reads the current status of the observer, page 214
BestObsErrStringGet ( )	Returns the protocol rule text for a specific error, page 216
BestObsStatusClear ( )	Clears the current status of the observer, page 218
BestObsRuleGet ( )	Clears the current status of the observer, page 217
BestObsRun ( )	Starts the protocol observer running, page 219

BestObsStop ( )	Stops the protocol observer, page 220

PCI Trace, Analyzer and External Trigger Functions

These functions control the PCI Static Logic Analyzer. They setup the trigger and storage qualilfier, start and stop the analyzer and to readout the result memory.

BestTracePropSet ( )	Sets the analyzer trace memory trigger mode, page 221
BestTracePattPropSet ( )	Sets the trigger and storage qualifier patterns, page 223
BestTraceDataGet ( )	Reads the current analyzer trace memory contents, page 224
BestTraceBitPosGet ( )	Reads the bit position of specific signal in the analyzer state line, page 225
BestTraceBytePerLineGet ( )	Reads the bytes per analyzer line for the BEST hardware you are using, page 227
BestTraceStatusGet ( )	Reads the current analyzer trace memory status, page 228
BestTraceRun ( )	Enables the analyzer trace memory for triggering, page 230
BestTraceStop ( )	Stops the current trace memory run, page 231
BestAnalyzerRun ( )	Starts the observer and enables trace memory, page 232
BestAnalyzerStop ( )	Stops the observer and trace memory acquisition, page 233

Port Programming Functions

These functions control onboard Static IO and the CPU Ports.

Static IO Port Functions	
BestStaticPropSet ( )	Sets the pin type of the Static IO port, page 234
BestStaticWrite ( )	Writes a byte value to the Static IO port, page 236
BestStaticRead ( )	Reads the current value of the Static IO port, page 237
BestStaticPinWrite ( )	Sets a specific Static IO output pin, page 238
CPU Port Functions	

BestCPUportPropSet ( )	Sets the mode, protocol type and RDY# signal, page 239
BestCPUportWrite ( )	Writes a word to the CPU port, page 241
BestCPUportRead ( )	Reads a word from the CPU port, page 242
BestCPUportIntrStatusGet ( )	Reads the interrupt status of the CPU port, page 243
BestCPUportIntrClear ( )	Clears the CPU port interrupts, page 244
BestCPUportRST ( )	Sets or resets the static CPU port reset signal, page 245

Configuration Space, BEST Status, and Interrupt Functions

These functions control configuration space registers and the onboard PCI Expansion ROM.

Configuration Space Functions	
BestConfRegSet ( )	Sets a register in the configuration space header, page 246 . This function will fail if it is called while a transaction is running
BestConfRegGet ( )	Reads a register in the configuration space header, page 247
BestConfRegMaskSet ( )	Sets the configuration access programming mask, page 248
BestConfRegMaskGet ( )	Sets the configuration access programming mask, page 249
Expansion ROM Functions	
BestExpRomByteWrite ( )	Writes a byte to the onboard PCI Expansion ROM, page 250
BestExpRomByteRead ( )	Reads a byte from the onboard PCI Expansion ROM, page 251
BEST Status Register Functions	
BestStatusRegGet ( )	Reads the onboard status register (not PCI Status Reg), page 252
BestStatusRegClear ( )	Clears the onboard status register, page 254
Interrupt Functions	
BestInterruptGenerate ( )	Generates an PCI interrrupt, page 255

Mailbox and Hex Display Functions

These functions control the mailbox and the onboard HEX display

Mailbox Functions	
BestMailboxSendRegWrite ( )	Writes a byte message to the send mailbox (through external intewr-face), page 256
BestMailboxReceiveRegRead ( )	Reads a message from the receive mailbox(through external intewr-face),, page 257
BestPCICfgMailboxSendRegWrite ( )	Writes a byte message to the send mailbox (through PCI config. space), page 258
BestPCICfgMailboxReceiveRegRead ( )	Reads a message from the receive mailbox (through PCI config. space), page 259
Hex Display Functions	
BestDisplayPropSet ( )	Sets the operating mode of the Hex display, page 260
BestDisplayWrite ( )	Writes a 8 bit value to the Hex display, page 261

Power Up Behavior Functions

These functions define the behavior of the board after power up. The onboard CPU runs a power up initialization function, which loads the defaults from the onboard EEPROM.

BestPowerUpPropSet ( )	Determines if the analyzer and observer run after power up, page 262
BestPowerUpPropGet ( )	Reads the current power up properties, page 264
BestAllPropStore ( )	Stores all board properties to the user default space, page 265
BestAllPropLoad ( )	Loads all board properties from user default space, page 266
BestAllPropDefaultLoad ( )	Loads all board properties with the factory defaults, page 267

Miscellaneous Functions

These functions control miscellaneous board functions

BestDummyRegWrite ( )	Writes to a non functional onboard register, page 268
BestDummyRegRead ( )	Reads from the above non functional onboard register, page 269
BestErrorStringGet ( )	Returns the error string for an error number, page 270
BestVersionGet ( )	Reads version numbers and dates from the hardware, page 271
BestSMReset ( )	Performs a board statemachine reset, page 273
BestBoardReset ( )	Performs an equivalent hardware reset, page 274
BestBoardPropSet ( )	Sets the type of PCI reset, page 275
BestBoardPropGet ( )	Reads the current type of PCI reset, page 277

Host to PCI Access Functions

These functions allow the direct access from the host to the address range of the system under test or to the internal PCI data memory of BEST via the controlling interface port.

BestHostSysMemAccessPrepare ( )	Sets parameters for SysMemFill and SysMemDump, page 278
BestHostSysMemFill ( )	Transfers host -> PCI memory transfer (mem_write bursts), page 279
BestHostSysMemDump ( )	Transfers PCI memory -> host transfer (mem_read bursts), page 281
BestHostIntMemFill ( )	Transfers host buffer -> internal memory, page 283
BestHostIntMemDump ( )	Transfers internal memory -> host buffer, page 285
BestHostPCIRegSet ( )	Sets a PCI register in any address space on the bus, page 286
BestHostPCIRegGet ( )	Reads a PCI register from any address space on the bus, page 288

Return Values

Return value	Description
B_E_OK	no error
B_E_ERROR	error transferring command
B_E_FUNC	functional onboard error
B_E_NO_HANDLE_LEFT	no handles left
B_E_HANDLE_NOT_OPEN	handle not in use
B_E_BAD_HANDLE	bad handle
B_E_NO_PCI_CLK	no or slow PCI clock
B_E_BAD_DECODER_NUMBER	no valid decoder number
B_E_BAUDRATE	could not set new baudrate
B_E_CPU_MISALIGNED	CPUport address is misaligned
B_E_FILE_OPEN	could not open file
B_E_BAD_FILE_FORMAT	error in file format
B_E_HOST_MEM_FULL	insufficient host memory
B_E_JEDEC_UES	could not include UES in JEDEC stream
B_E_WRONG_PARAMETER	wrong parameter value in function call
B_E_RS232_OPEN	could not open RS232 port
B_E_NOT_CONNECTED	port is not connected
B_E_WRONG_PORT	invalid port specified
B_E_CONNECT	unable to connect, another host is probably already connected
B_E_PCI_OPEN	could not open Best PCI driver
B_E_CHECK	error while checking connection
B_E_PARALLEL_OPEN	could not open BEST EPP port driver

B_E_CONF_REG	could not write to config space or value is invalid for the specified register
B_E_MASK_REG	could not write config mask or value is invalid for the specified register
B_E_DEC_CHECK	unknown decoder check error. Maybe there is no PCI clock or master is running.
B_E_CORE_PORT	Must use RS232 port with 9600 baud to switch to CORE-BIOS
B_E_VERSION_CHECK	could not find valid version information in file
B_E_IO_OPEN	could not open BEST IO port driver
B_E_PROG_DEC_ENABLE	BEST IO programming decoder not enabled, or could not read from config space
B_E_WRONG_PROP	unknown or invalid property specified
B_E_NO_BEST_PCI_DEVICE_FOUND	specified device not found on PCI bus
B_E_TEST_NO_DECODER	first decoder not enabled
B_E_INVALID_OBS_RULE	observer rule unknown or not implemented
B_E_CAPI_VERSION_CHECK	this C-API version cannot work with the detected board bios version
B_E_NO_DWORD_BOUNDARY	specified address must be on a DWORD boundary
B_E_NO_WORD_BOUNDARY	specified address must be on a WORD boundary
B_E_DEC_SIZE_BASE_MISMATCH	decoder size and base address mismatch
B_E_DEC_INVALID_SIZE	invalid size specified for decoder
B_E_DEC_INVALID_MODE	invalid mode specified for decoder
B_E_DEC_INVALID_DECODER	unimplemented decoder
B_E_PCI_OPEN	error while addressing config space
B_E_BASE_WRITE	error while writing base address into HW
B_E_DEC_BASE_NOT_0	base address must be 0 when size is 0
B_E_BOARD_RESET	could not reconnect after board reset

B_E_MASTER_ABORT	master abort occurred and there was no target response
B_E_DEC_ROM_SIZE_0_ENABLE	generic ROM enable is active but size = 0

Chapter 9 Control and Programming Interfaces

This chapter describes the 3 programming interfaces.

This chapter contains the following sections:

Overview

All exerciser and analyzer functionality is programmed by write accesses to registers of an onboard, internal register space.The state of these registers define exactly the state and behavior of the board.

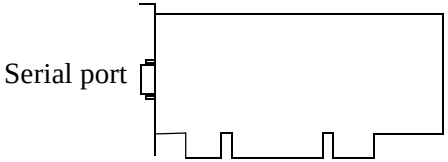
This register space is not intended to be public, but it helps to be aware of the accessing mechanisms.

These registers can be accessed from the following three interfaces:

- RS232
- Parallel bidirectional Centronics Interface (IEEE1284 C , Enhanced EPP 1.9 protocol)
- PCI (through programming registers in the user configuration space and I/O space).

Because it is possible to control the card by more than one interface during one session, a register space lock mechanism implemented, which ensures register space consistency for a given number of accesses from one port, and to delay the access from any other port, until the resource is free again.

RS232 Controlling Interface

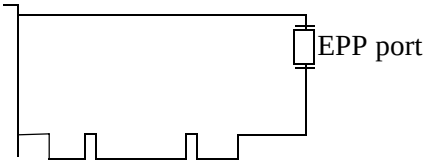


The RS232 serial interface provides a standard interface for connecting a PC or notebook external controller. An 9 pin to 9pin RS232 cable is delivered as standard with each Analyzer and Exerciser card. If a 25 pin to 9 pin adapter is used it must connect the hanshaking lines (DTR, DSR, RTS, CTS).

The following baud rates are supported: 9600, 19200, 38400, 57600

Using: 8 bit data, 1 stop bit and no parity

Fast Host (Parallel EPP) Controlling Interface



The parallel interface provides a high speed host interface for applications that require movement of large data blocks to or from the card, and also provides a complete programming interface from the external host.

Uses EPP 1.9 protocol

Net transfer rate from onboard memory to host = 150kByte/s.

An ISA based interface card and a cable connecting the parallel port connector of this card to the IEEE 1284C connector of the exerciser is shipped as product option 003.

The parallel port on your PC may be used if it supports the EPP 1.9 protocol. Before using the parallel interface, ensure that the parallel port is in EPP mode (refer to your PC's documentation for information on how to do this).

PCI Controlling Interface

The DUT itself can be used as the controlling host. In this case the DUT must be an IBM Compatible PC running DOS, and have a PCI compliant Bios, or NT. The controlling accesses are done by a set of programming registers in the user configuration space of the card. Normally you don’t have to care about this, because the PCI driver is delivered with the software.

If you want to use the tool through PCI on a non IBM compatible platform, you must write a driver. This procedure is described in *Porting the PCI interface driver to a UNIX platform on page 94*.

Programming Register Layout

The PCI programming registers can be located in configuration space. The configuration space decoder must be enabled for controlling through PCI bus. If controlling through PCI you will see accesses to the following configuration space registers for host control purposes:

31:24	23:16	15:8	7:0	Config Offset
			mailbox status	50\h
			mailbox register	4C\h
		connect status	connect command	48\h
			status	44\h
register block transfer parameters				40\h

Chapter 10 DUT and Instrument Interfaces

This chapter describes the 3 possible programming interfaces.

This chapter contains the following sections:

Overview

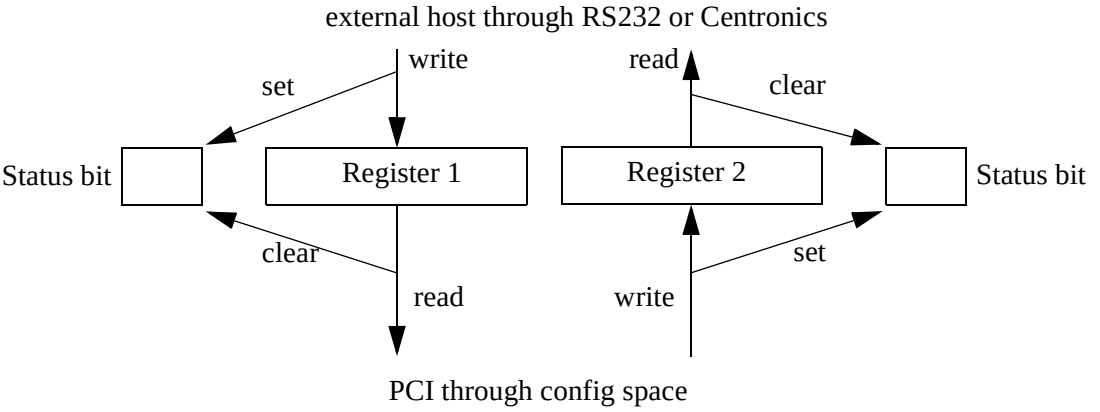
This chapter describes the interfaces between the DUT and the controlling host. These interfaces allow communication between the host and the DUT CPU and allow further analysis of data on the PCI bus using a Logic Analyzer. The CPU port and static I/O port provide a direct link to the DUT and may be used to initialize the DUT. The mailbox registers allow the host to communicate with the DUT via the PCI bus. The external trigger inputs allow the anan-lyzer to be triggered by signals which are not available on the PCI bus.

This chapter describes the operation of these interfaces and how they may be used. The pinout of each of the physical interfaces is described.

Mailbox Registers

The card provides two mailbox registers, which are part of the configuration space, for communication between programs running on the external host and the program executed by the DUT CPU. The mailbox registers can be accessed, using the Mailbox Register Access Functions of the C-API library. See “Mailbox and Hex Display Functions” on page 296.

Physically, the registers are built up as two unidirectional byte registers, each with a corresponding status bit.



From the external host register 1 is the send register and register 2 is the receive register. From PCI the opposite is true. A write to the send register automatically sets the send status bit. A read from the receive register clears the corresponding receive status bit.

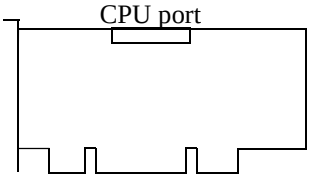
Two groups of functions are provided:

- *BestMailboxSendRegWrite ( ) on page 256* and *BestMailboxReceiveRegRead ( ) on page 257* provide access to the mailbox registers through the external interfaces (*RS232 Controlling Interface on page 303* and *Fast Host (Parallel EPP) Controlling Interface on page 304*).
- *BestPCICfgMailboxSendRegWrite ( ) on page 258* and *BestPCICfgMailboxReceiveRegRead ( ) on page 259* provide access to the mailbox registers through the PCI interface (*PCI Controlling Interface on page 305*).

Offset Config	Offset IO	Bits	Type	Access	Meaning
4C\h	0C\h	[7:0]	RW	read	Reads the mailbox register.
				write	Writes to the mailbox register.
4D\h	0D\h	[1]	RO	0	Status Reg, allowed to write to the send register
				1	Status Reg, last data not yet fetched by the host.
		[0]	RO	0	Status Reg, no valid data in the receive register
				1	Status Reg, valid data in the receive register

If the card is programmed through PCI, accesses to this register can be observed on the PCI bus.

CPU Port



Overview

The CPU port is a simple parallel programming interface. It can be used in applications, where a device under test needs initialization of a set of registers before the tests can be performed.

The CPU port provides the address, data bus, control, two chip selects and two byte enables. With these signals it is possible to access different devices on one board, without the need for external decoding hardware.

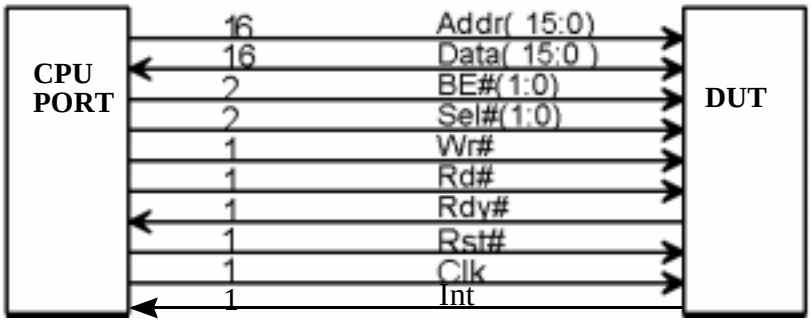
As the card is designed to be a PCI test tool, it makes sense to use it in applications where both ports play a role. (See “Port Programming Functions” on page 294.)

C-Lib Functions

The CPU port can be accessed from all 3 controlling port types, using a few C-API function calls.

Intel Compatible Interface

Signals



Signal	Description
Addr (15:0)	Address Bus AD(15) = MSB
Data (15:0)	Data Bus D (15) = MSB
BE#(1:0)	Active Low byte enables, specifying the byte lane carrying meaningful data.
Sel# (1:0)	Active Low Chip Select Signals
Wr#	Active Low Write Enable signal
Rd#	Active Low Read Enable Signal

Signal	Description
Rdy#	Rdy is low, when data has been transferred
Rst#	Active Low reset output for CPU port
Clk	CPU clock of onboard Controller system (16MHz)
Int	Interrupt line

Chip Selects the two Chip Select lines can be set by parameter 'device_num' of the *BestCPUportWrite ( )* or *BestCPUportRead ( )* function. This feature enables you to address up to 2 external devices without the need of separate decoding logic, providing a 64kByte address space for each Chip select

device_num	Sel 1#	Sel 0#
0	1	0
1	0	1

Byte and Word Accesses the CPU port allows byte and word accesses. The following combinations between AD[0], BE[1:0] and D[15:0] are used.

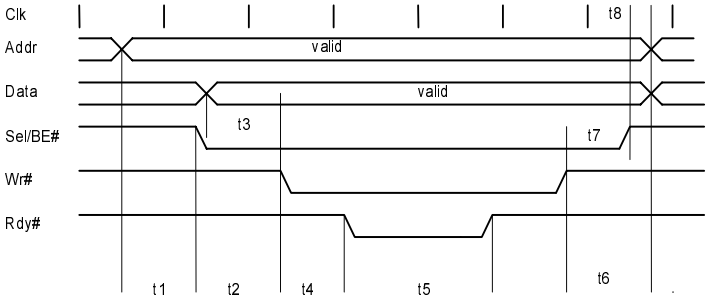
BE1 indicates that D[15:8] carries valid data. BE0 asserted means that D[7:0] carries meaningful data.

Size	AD[0]	BE1	BE0	D[15:8]	D[7:0]
Byte	0	1	0	-	Byte
Byte	1	0	1	Byte	-
Word	0	0	0	Hi_Byte	Lo_Byte
Word	1			not allowed	

The transfer size is controlled using the 'size' parameter of the CPUPort Functions. Word accesses to non-word address (AD[0] = 1) will be terminated with an transfer error.

Reset the RST# signal of the CPUport can be asserted and deasserted using the *BestCPUportRST ( )* function.

Write Timing the control signals are generated by a synchronous statemachine, controlled from the negative edge of the CPU clock. This clock is also available on the CPU port connector.



Read Timing Read timing

Table 18

	min	max	unit
t1	60		ns
t2	60		ns
t3	50		ns
t4	0		ns
t5	130		ns
t6	60		ns
t7	60		ns
t8	5		ns
t9	0		ns
t10	0		ns

Interrupts The interrupt line is edge-triggered. *BestCPUportIntrStatusGet ( ) on page 243* reads the interrupt latch and *BestCPUportIntrClear ( ) on page 244* clears it.

There is an automatic_ready mode, which can be enabled during initialization of the CPUport. With this mode, the Rdy# signal will be generated internally for 300ns (see *BestCPUportPropSet ( ) on page 239*).

CPU port pinout

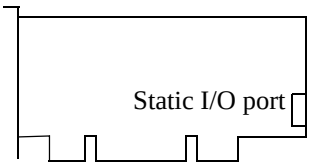
Pin#	Signal	Pin#	Signal	Pin#	Signal	Pin#	Signal
1	Wr#	2	GND	41	A(13)	42	A(14)
3	Int#	4	GND	43	A(15)	44	GND
5	Rd#	6	GND	45	D(0)	46	GND
7	Rst#	8	GND	47	D(1)	48	GND
9	Sel(0)#	10	GND	49	D(2)	50	GND

11	Sel(1)#	12	GND	51	D(3)	52	GND
13	BE(0)#	14	GND	53	D(4)	54	GND
15	Rdy#	16	GND	55	D(5)	56	GND
17	A(0)	18	GND	57	D(6)	58	GND
19	A(1)	20	GND	59	D(7)	60	GND
21	A(2)	22	GND	61	BE (1)#	62	GND
23	A(3)	24	GND	63	D(8)	64	GND
25	A(4)	26	GND	65	D(9)	66	GND
27	A(5)	28	GND	67	D(10)	68	GND
29	A(6)	30	GND	69	D(11)	70	GND
31	A(7)	32	GND	71	D(12)	72	GND
33	A(8)	34	GND	73	D(13)	74	GND
35	A(9)	36	GND	75	D(14)	76	GND
37	A(10)	38	A(12)	77	D(15)	78	GND
39	A(11)	40	GND	79	Clk	80	GND

Signal and ground lines are arranged alternately on the flatcable to ensure optimal signal performance. The connector used is an 80-pin finepitch flatcable connector.with the following details:

AMP AMPMODU System50
Manufacturer Part#: 104549-9 (80 positions)

Static I/O Port



Static I/O Signals

8 independent static i/o signals provide a means of controlling up to 8 separate lines with programmable electrical characteristics (Open Drain, Totem Pole, Input).

The output characteristic and the logical state of the static i/o outputs can be programmed by C-API functions (See “Port Programming Functions” on page 294.)

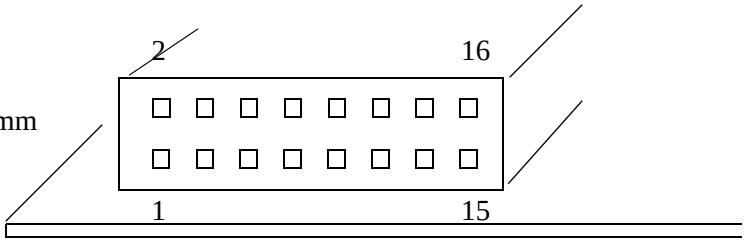
The state of the 8 signals can also be read in by a API function.

Connector 16 Pin flatcable connector. This connector type allows you to attach a grabber cable from the logic analyzer or flying leads to the DUT.

Manufacturer AMP

Drawing Front View of the onboard connector

Pin distance 2.54mm * 2.54mm

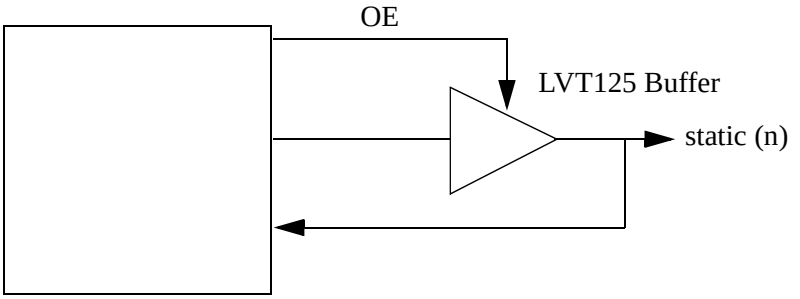


Pin Layout GND lines and signal lines are arranged alternately to achieve better switching characteristics.

Pin #	Signal	Pin #	Signal
1	GND	2	Static (0)
3	GND	4	Static (1)
5	GND	6	Static (7)
7	GND	8	Static (7)
9	GND	10	Static (7)
11	GND	12	Static (7)
13	GND	14	Static (7)
15	GND	16	Static (7)

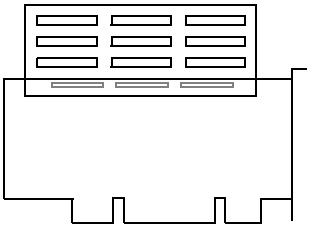
Drivers Each Pin is driven by an LVT125 Open Drain Buffer. This 3.3V driver allows you to connect to both, 5V and 3.3V systems. To get more detailed information about electrical specifications please refer to the vendor's data book.

Each static i/o signal is connected directly to a separate pin of the device bus interface FPGA, and can only be used to read back the current value, see C-API function *BestStaticRead ( ) on page 237*



NOTE: During reset all Static IO pins are pulled to +5V.

Option 003/004 Logic Analyzer Adapter



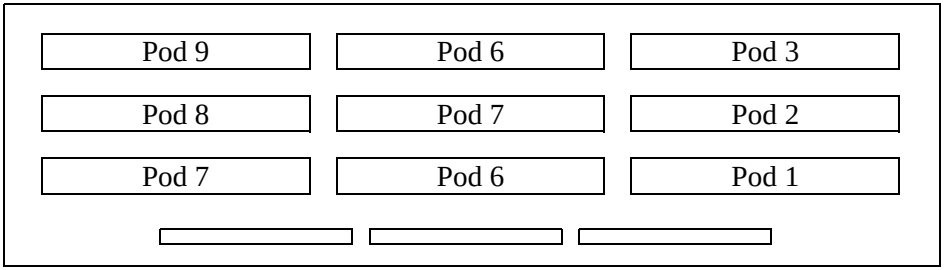
The option 003 adapter allows you to connect an HP Logic Analyzer directly to the card and the option 004 adapter allows you to connect any other logic analyzer to the card. In addition to the PCI bus signals this interface provides the internal state of the of the exerciser and anlyzer.

NOTE: The signals available at this adapter are delayed by several clock cycles.

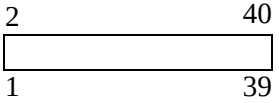
Logic Analyzer Settings for 16550 HP analyzers are shipped as part of the software release. These can be found in the installation \la directory.

LA connectors

The Logic Analyzer Pods are connected as follows:



The connector pins are numbered as follows:



Pod 1

Pin	LA	Signal	Pin	LA	Signal
1	nc	nc	2	GND	GND
3	CLK	LA_PCI_CLK	4	GND	GND
5	nc	nc	6	GND	GND
7	D15	AD32[15]	8	GND	GND
9	D14	AD32[14]	10	GND	GND
11	D13	AD32[13]	12	GND	GND

13	D12	AD32[12]	14	GND	GND
15	D11	AD32[11]	16	GND	GND
17	D10	AD32[10]	18	GND	GND
19	D9	AD32[9]	20	GND	GND
21	D8	AD32[8]	22	GND	GND
23	D7	AD32[7]	24	GND	GND
25	D6	AD32[6]	26	GND	GND
27	D5	AD32[5]	28	GND	GND
29	D4	AD32[4]	30	GND	GND
31	D3	AD32[3]	32	GND	GND
33	D2	AD32[2]	34	GND	GND
35	D1	AD32[1]	36	GND	GND
37	D0	AD32[0]	38	GND	GND
39	nc	nc	40	GND	GND

Pod 2

Pin	LA	Signal	Pin	LA	Signal
1	nc	nc	2	GND	GND
3	CLK	DBI_CPU_CLK	4	GND	GND
5	nc	nc	6	GND	GND
7	D15	AD32[31]	8	GND	GND
9	D14	AD32[30]	10	GND	GND
11	D13	AD32[29]	12	GND	GND
13	D12	AD32[28]	14	GND	GND
15	D11	AD32[27]	16	GND	GND
17	D10	AD32[26]	18	GND	GND
19	D9	AD32[25]	20	GND	GND
21	D8	AD32[24]	22	GND	GND
23	D7	AD32[23]	24	GND	GND
25	D6	AD32[22]	26	GND	GND
27	D5	AD32[21]	28	GND	GND

29	D4	AD32[20]	30	GND	GND
31	D3	AD32[19]	32	GND	GND
33	D2	AD32[18]	34	GND	GND
35	D1	AD32[17]	36	GND	GND
37	D0	AD32[16]	38	GND	GND
39	nc	nc	40	GND	GND

Pod 3

Pin	LA	Signal	Pin	LA	Signal
1	nc	nc	2	GND	GND
3	CLK	nc	4	GND	GND
5	nc	nc	6	GND	GND
7	D15	SERR	8	GND	GND
9	D14	PERR	10	GND	GND
11	D13	IDSEL	12	GND	GND
13	D12	STOP	14	GND	GND
15	D11	DEVSEL	16	GND	GND
17	D10	IRDY	18	GND	GND
19	D9	TRDY	20	GND	GND
21	D8	FRAME	22	GND	GND
23	D7	RST	24	GND	GND
25	D6	BSTATE[2]	26	GND	GND
27	D5	BSTATE[1]	28	GND	GND
29	D4	BSTATE[0]	30	GND	GND
31	D3	CBE[3]	32	GND	GND
33	D2	CBE[2]	34	GND	GND
35	D1	CBE[1]	36	GND	GND
37	D0	CBE[0]	38	GND	GND
39	nc	nc	40	GND	GND

Pod 4

Pin	LA	Signal	Pin	LA	Signal

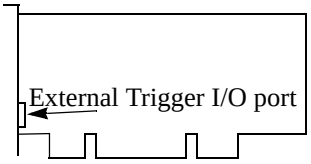
1	nc	nc	2	GND	GND
3	CLK	nc	4	GND	GND
5	nc	nc	6	GND	GND
7	D15	INTC	8	GND	GND
9	D14	INTB	10	GND	GND
11	D13	INTA	12	GND	GND
13	D12	TRIG3	14	GND	GND
15	D11	TRIG2	16	GND	GND
17	D10	TRIG1	18	GND	GND
19	D9	TRIG0	20	GND	GND
21	D8	SBO	22	GND	GND
23	D7	SDONE	24	GND	GND
25	D6	BERR	26	GND	GND
27	D5	PAR	28	GND	GND
29	D4	M_ACT	30	GND	GND
31	D3	T_ACT	32	GND	GND
33	D2	LOCK	34	GND	GND
35	D1	GNT	36	GND	GND
37	D0	REQ	38	GND	GND
39	nc	nc	40	GND	GND

Pod 5

Pin	LA	Signal	Pin	LA	Signal
1	nc	nc	2	GND	GND
3	CLK	nc	4	GND	GND
5	nc	nc	6	GND	GND
7	D15	nc	8	GND	GND
9	D14	nc	10	GND	GND
11	D13	DEVSEL_OE	12	GND	GND
13	D12	FRAME_OE	14	GND	GND
15	D11	PERR_OE	16	GND	GND
17	D10	IRDY_OE	18	GND	GND

19	D9	CBE_OE	20	GND	GND
21	D8	AD_OE	22	GND	GND
23	D7	nc	24	GND	GND
25	D6	nc	26	GND	GND
27	D5	nc	28	GND	GND
29	D4	nc	30	GND	GND
31	D3	SQ	32	GND	GND
33	D2	M_LOCK	34	GND	GND
35	D1	T_LOCK	36	GND	GND
37	D0	INTD	38	GND	GND
39	nc	nc	40	GND	GND

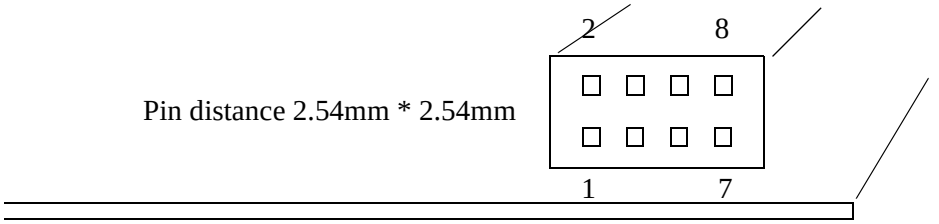
External Trigger I/O



These signals allow you to input 3 external trigger signals to the onboard analyzer.

One of the pins can also be used as a trigger output, which provides triggering for an external logic analyzer or scope. This connector is located next to the serial port.

Drawing



Pin Layout

Pin	Signal	Pin	Signal	type
1	GND	2	TRIGGER_0[a]	I/O
3	GND	4	TRIGGER_1	input only
5	GND	6	TRIGGER_2	input only
7	GND	8	TRIGGER_3	input only

a. This pin cannot be used as input. It currently carries the external trigger output, which is also used by the internal pattern terms.

Chapter 11 Miscellaneous Interfaces

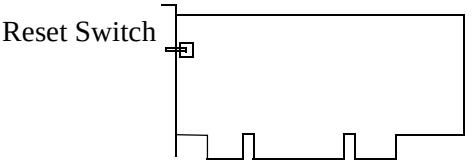
This chapter describes the interfaces between the.

This chapter contains the following sections:

Overview

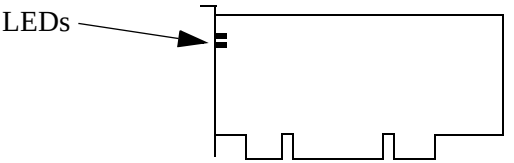
The exerciser/analyzer card has several interfaces which indicate the status of the card. The card may also be reset and the power source selected.

Reset Switch

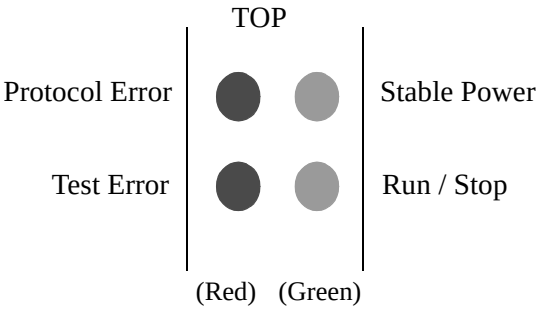


The reset switch on the card front panel performs a hard reset to the complete board. This resets all statemachines, exerciser and analyzer registers.

LEDs

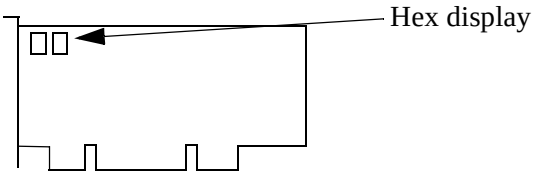


The LEDs provide minimum information on the current system/card status:.



Test error illuminates if a data compare error occurs. Protocol Error indicates that the protocol observer detected a prototcol violation.

Hex Display



The hex display can be used in two different modes:

- the default mode, displaying the error number of the first detected PCI protocol error (in hexadecimal - see *Protocol Observer on page 141* for descriptions of the protocol violations checked)
- user mode when using the C-API, display your own test relevant information on the display. See *BestDisplayWrite ( ) on page 261*.

A display mode may be selected using *BestDisplayPropSet ( ) on page 260*.

Independently of the selected mode, the hex display is also used to indicate internal error conditions on the HP E2925A card. Those error conditions are indicated by a flashing display (rather than a steady display for the protocol errors). Table 19 shows the possible codes.

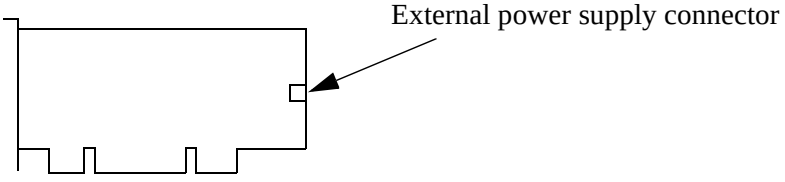
Table 19 Meaning of Hex display error

Number on hex display	Appearance	Description
??	flashes once, approx. 2 sec after power up	indicates the firmware version number that isinstalled on the card. For the current software revision (3.21.00) this is a 03.
AA	flashes three times approx. 2 sec after power up.	indicates that the Altera FPGA's could not be configured. Most likely this is due to a missing +5V supply on the PCI connector.

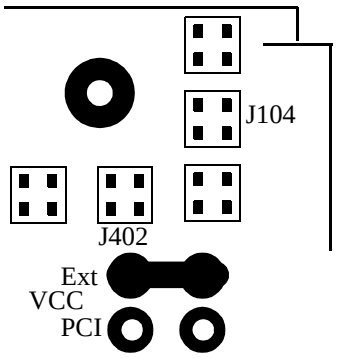
Table 19 Meaning of Hex display error

Number on hex display	Appearance	Description
CC	flashes three times,approx. 4 sec after power up	Indicates that the PCI CLK signal is not active. The HP E2925A does not properly configure itself in this case. Please make sure that the PCI clock signal becomes active no more than 3 seconds after pow-erup.
EE	flashes three times,approx. 2 sec afterpower up	Indicates that the data in the on-board non-volatile RAM is corrupt and will be reset to factory defaults.
FF	continuously on	Indicates an internal error in the protocol observer. To resolve, press the red reset button on the card or cycle power. If the problem persists, please contact technical support.
??	flashes infinitely, anytime during operation	Indicates an internal error of the on-board CPU. To resolve, press the red reset button on the card or cycle power. If the problem persists, please contac technical support.

External Power



Connector to be used with the optional power supply for applications where the power cannot be drawn from the PCI connector.



It delivers +5V for the core logic on the board.

The power supply used by the E2925A card is determined by the VCC jumper. Set this jumper to ‘Ext’ to enable the board to use the external power supply..

The PCI drivers are always powered from the PCI connector.

Index

Index

Index