ALPHA DATA

ADM-XRC Gen 3 SDK User Guide

Revision: 1.1
Date: 26th July 2010

	Head Office	US Office
Address	4 West Silvermills Lane, Edinburgh, EH3 5BD, UK	2570 North First Street, Suite 440 San Jose, CA 95131
Telephone	+44 131 558 2600	(408) 467 5076 General (408) 916 5713 Sales
Fax	+44 131 558 2700	(408) 436 5524 (866) 820 9956 toll free
email	sales@alpha-data.com	sales@alpha-data.com
website	http://www.alpha-data.com	http://www.alpha-data.com

Table Of Contents

Tables

Figures

1 Introduction

This document describes the ADM-XRC Gen 3 Software Development Kit (SDK), which provides resources for developers working with the third generation of reconfigurable computing hardware from Alpha Data. The key features of the SDK are:

- Example applications that use the ADMXRC3 API.
- Example HDL FPGA designs that target third generation Alpha Data hardware such as the ADM-XRC-6TL. These designs are built from a number of HDL components that are also provided in this SDK.
- Utilities for working with third generation Alpha Data hardware.

1.1 Supported operating systems

This SDK supports the following operating systems:

- Windows NT-based operating systems beginning with Windows 2000. Both 32-bit and 64-bit editions are supported.
- Linux distributions running a 2.6.x kernel.

1.2 Supported Alpha Data hardware

The example applications and HDL code in this SDK support the following models in Alpha Data's range of reconfigurable computing hardware:

- ADM-XRC-6TL
- ADM-XRC-6T1

1.3 Installation

1.3.1 Installation in Windows

The default installation location depends upon whether the operating system is a 32-bit or 64-bit edition of Windows:

- **%ProgramFiles%\ADMXRCG3SDK-*release*** in 32-bit editions of Windows.
- **%ProgramFiles(x86)%\ADMXRCG3SDK-*release*** in 64-bit editions of Windows.

where ***release*** is the release number of this package.

During installation, the installer automatically creates an environment variable **ADMXRC3_SDK** that points to where the SDK is installed. Certain example applications use this environment variable to locate FPGA bitstream (.BIT) files. A user need not manually set this variable, but if using several versions of the SDK, it can be set manually according to which version of the SDK is in use.

1.3.2 Installation in Linux

This SDK is supplied as a tarball (.tar.gz extension) that should normally be extracted to the ***/opt*** directory, which places the root of the SDK at**/opt/admxrcg3sdk-*release*** directory, where ***release*** is the release number of this package.

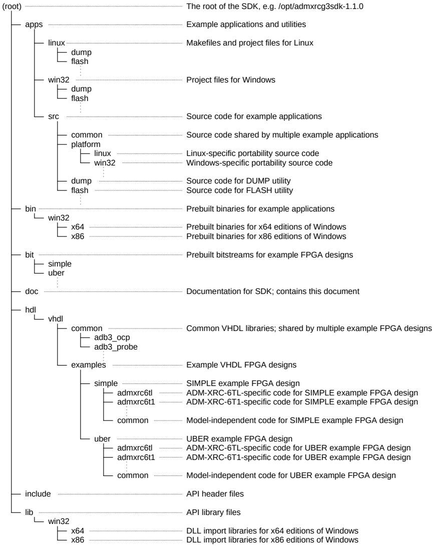
After installation, an environment variable **ADMXRC3_SDK** must be defined that points to where the SDK is installed. Certain example applications use this environment variable to locate FPGA bitstream (.BIT) files. A convenient way to permanently define this variable for a given user is to add the following to the user's **.bash_profile**:

```
ADMXRC3_SDK=/opt/admxrcg3sdk-<i>release</i>
export ADMXRC3_SDK
```

1.3.3 Installation in VxWorks

Since VxWorks normally requires a Windows, Linux or UNIX host, this SDK must be installed on a Windows or Linux host as described in **Section 1.3.1, "Installation in Windows"** or **Section 1.3.2, "Installation in Linux"**.

1.4 Structure of this SDK



```
(root) ........................ The root of the SDK, e.g. /opt/admxrcg3sdk-1.1.0
  apps ........................ Example applications and utilities
    linux ..................... Makefiles and project files for Linux
      dump
      flash
      :
    win32 ..................... Project files for Windows
      dump
      flash
      :
    src ....................... Source code for example applications
      common .................. Source code shared by multiple example applications
      platform
        linux ................. Linux-specific portability source code
        win32 ................. Windows-specific portability source code
        :
      dump .................... Source code for DUMP utility
      flash ................... Source code for FLASH utility
      :
  bin ......................... Prebuilt binaries for example applications
    win32
      x64 ..................... Prebuilt binaries for x64 editions of Windows
      x86 ..................... Prebuilt binaries for x86 editions of Windows
  bit ......................... Prebuilt bitstreams for example FPGA designs
    simple
    uber
    :
  doc ......................... Documentation for SDK; contains this document
  hdl
    vhdl
      common .................. Common VHDL libraries; shared by multiple example FPGA designs
        adb3_ocp
        adb3_probe
        :
      examples ................ Example VHDL FPGA designs
        simple ................ SIMPLE example FPGA design
          admxrc6tl ........... ADM-XRC-6TL-specific code for SIMPLE example FPGA design
          admxrc6t1 ........... ADM-XRC-6T1-specific code for SIMPLE example FPGA design
          :
          common .............. Model-independent code for SIMPLE example FPGA design
        uber .................. UBER example FPGA design
          admxrc6tl ........... ADM-XRC-6TL-specific code for UBER example FPGA design
          admxrc6t1 ........... ADM-XRC-6T1-specific code for UBER example FPGA design
          :
          common .............. Model-independent code for UBER example FPGA design
  include ..................... API header files
  lib ......................... API library files
    win32
      x64 ..................... DLL import libraries for x64 editions of Windows
      x86 ..................... DLL import libraries for x86 editions of Windows
```

Figure 1: Structure of the ADM-XRC Gen 3 SDK

2 Getting started

2.1 Getting started in Windows 2000 / XP / Server 2003

Note: This section also applies to Windows Vista and later when User Account Control (UAC) is disabled.

This section describes how to run a basic confidence test on Alpha Data hardware, in Windows 2000 / XP / Server 2003. This confidence test assumes the following:

1. All features of the SDK were installed, as described in **Section 1.3, "Installation"**.
2. Any model from Alpha Data's reconfigurable computing range that is supported by this SDK is installed in the machine. For a list of hardware supported, refer to section **Section 1.2, "Supported Alpha Data hardware"**.
3. The ADB3 driver is installed. The ADB3 driver for Windows is available from Alpha Data's public FTP site: **ftp://ftp.alpha-data.com/pub/admxrcg3/windows**.
4. You are logged on as a user that is a member of the Administrators group.

First, start an SDK command prompt by clicking on the 'SDK Command Prompt' shortcut from the 'ADM-XRC Gen 3 SDK' group on the Windows start menu. This command prompt automatically starts with the working directory set to the **bin/win32/x86/** folder of the SDK and also ensures that the **ADMXRC3_SDK** environment variable is set correctly.

Next, run the **info** utility. The output looks like this:

```
API information
API library version           1.1.0
Driver version                1.1.0

Card information
Model                         ADM-XRC-6TL
Serial number                 101(0x65)
Number of programmable clocks 1
Number of DMA channels        1
Number of target FPGAs        1
Number of local bus windows   4
Number of sensors             40
Number of I/O module sites    1
Number of local bus windows   4
Number of memory banks        4
Bank presence bitmap          0xF

Target FPGA information
FPGA 0                        xc6vlx240tff1759

Memory bank information
Bank 0                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 1                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 2                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 3                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1

Local bus window information
Window 0  (Target FPGA 0 pre  Bus base     0xF5400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
  Window 1  (Target FPGA 0 non Bus base     0xFAC00000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
  Window 2  (ADM-XRC-6TL-speci Bus base     0xFAAFF000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
  Window 3  (ADB3 bridge regis Bus base     0xFAAFE000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
```

Now run the **simple** example application. It prompts the user to enter hexadecimal values (up to 32 bits), and displays these value nibble-reversed. The nibble-reversal is performed by the target FPGA. Hitting CTRL-Z exits this example. The output looks like this:

```
====================
Enter values for I/O
(use 55AA to exit)
====================
1234abcd
OUT = 0x1234abcd, IN = 0xdcba4321
55aa
OUT = 0x000055aa, IN = 0xaa550000
```

If everything works as described above, then the hardware, driver and SDK are all correctly installed and substantially working. Possible next steps are:

- Experiment with modifying and rebuilding the **simple** example application in order to become familiar with the basics of the ADMXRC3 API.
- Experiment with modifying and rebuilding the **simple** example FPGA design in order to become familiar with creating FPGA designs for Alpha Data hardware.

2.2 Getting started in Windows Vista and later

Note: If User Account Control is disabled, please refer instead to the instructions in **Section 2.1, "Getting started in Windows 2000 / XP / Server 2003"**.

This section describes how to run a basic confidence test on Alpha Data hardware, in versions of Windows that have User Account Control (UAC) such as Windows Vista and later. This confidence test assumes the following:

1. All features of the SDK were installed, as described in **Section 1.3, "Installation"**.
2. Any model from Alpha Data's reconfigurable computing range that is supported by this SDK is installed in the machine. For a list of hardware supported, refer to section **Section 1.2, "Supported Alpha Data hardware"**.
3. The ADB3 driver is installed. The ADB3 driver for Windows is available from Alpha Data's public FTP site: **ftp://ftp.alpha-data.com/pub/admxrcg3/windows**.
4. You are logged on as a user that is a member of the Administrators group.

Because of User Account Control (UAC), it is not possible to make use of the 'SDK Command Prompt' shortcut that is installed along with the SDK. Instead, start a command prompt by right-clicking on the 'Command Prompt' shortcut in the 'Accessories' program group and selecting **'Run as administrator'**. This will typically incur a UAC confirmation prompt. Then, enter the following command (do not omit the double quotes):

```
"%admxrc3_sdk3%\env"
```

This executes the **env.bat** batch file, which sets up the environment and changes to the folder containing the prebuilt example application binaries. In order for this to work correctly, the **ADMXRC3_SDK** system environment variable must be correctly defined. The installer normally sets this variable, but if not, it must be set as a **system** environment variable to point to where the SDK is installed, using the Windows Control Panel.

Next, run the **info** utility. The output looks like this:

```
API information
API library version           1.1.0
Driver version                1.1.0

Card information
Model                         ADM-XRC-6TL
Serial number                 101(0x65)
Number of programmable clocks 1
Number of DMA channels        1
Number of target FPGAs        1
Number of local bus windows   4
Number of sensors             40
Number of I/O module sites    1
Number of local bus windows   4
Number of memory banks        4
Bank presence bitmap          0xF

Target FPGA information
FPGA 0                        xc6vlx240tff1759

Memory bank information
Bank 0                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 1                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 2                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 3                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1

Local bus window information
Window 0  (Target FPGA 0 pre  Bus base      0xF5400000 size 0x400000
                              Local base    0x0 size 0x400000
                              Virtual size  0x400000
Window 1  (Target FPGA 0 non  Bus base      0xFAC00000 size 0x400000
                              Local base    0x0 size 0x400000
                              Virtual size  0x400000
Window 2  (ADM-XRC-6TL-speci  Bus base      0xFAAFF000 size 0x1000
                              Local base    0x0 size 0x0
                              Virtual size  0x1000
Window 3  (ADB3 bridge regis  Bus base      0xFAAFE000 size 0x1000
                              Local base    0x0 size 0x0
                              Virtual size  0x1000
```

Now run the **simple** example application. It prompts the user to enter hexadecimal values (up to 32 bits), and displays these value nibble-reversed. The nibble-reversal is performed by the target FPGA. Hitting CTRL-Z exits this example. The output looks like this:

```
===================
Enter values for I/O
(use 55AA to exit)
===================
1234abcd
OUT = 0x1234abcd, IN = 0xdcba4321
55aa
```

```
OUT = 0x000055aa, IN = 0xaa550000
```

If everything works as described above, then the hardware, driver and SDK are all correctly installed and substantially working. Possible next steps are:

- Make a copy of the SDK in your own filespace, and use the copy to experiment with modifying and rebuilding the **simple** example application in order to become familiar with the basics of the ADMXRC3 API.
- Make a copy of the SDK in your own filespace, and use the copy to experiment with modifying and rebuilding the **simple** example FPGA design in order to become familiar with creating FPGA designs for Alpha Data hardware.

2.3 Getting started in Linux

This section describes how to run a basic confidence test on Alpha Data hardware, in Linux. This confidence test assumes the following:

1. This SDK is installed as described in **Section 1.3, "Installation"**, and the **ADMXRC3_SDK** environment variable is set to point to where the SDK has been installed.
2. Any model from Alpha Data's reconfigurable computing range that is supported by this SDK is installed in the machine. For a list of hardware supported, refer to section **Section 1.2, "Supported Alpha Data hardware"**.
3. The ADB3 driver is installed. The ADB3 driver for Linux is available from Alpha Data's public FTP site: **ftp://ftp.alpha-data.com/pub/admxrcg3/linux**.

Note: In the following text, it is assumed that it is possible to log in as 'root'. If a Linux distribution is used where users are expected to use 'sudo' rather than logging in as root, then in all of the following instructions, commands should be prefixed with 'sudo' so that the effect is the same as 'su' to 'root'.

Log in as root (if possible), change directory to where the SDK has been installed, and then run the **configure** script:

```
$ cd $ADMXRC3_SDK
$ ./configure
```

This detects certain features of the operating system environment so that the example applications can be built. Next, change directory to the Linux application directory:

```
$ cd apps/linux
$ make clean all
```

Having built the example applications, run the **info** utility:

```
$ info/info
```

The output looks like this:

```
API information
API library version           1.1.0
Driver version                1.1.0

Card information
Model                         ADM-XRC-6TL
Serial number                 101(0x65)
Number of programmable clocks 1
Number of DMA channels        1
Number of target FPGAs        1
Number of local bus windows   4
Number of sensors             40
Number of I/O module sites    1
Number of local bus windows   4
Number of memory banks        4
Bank presence bitmap          0xF
```

```
Target FPGA information
FPGA 0                        xc6vlx240tff1759
Memory bank information
Bank 0                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 1                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 2                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 3                        SDRAM, DDR3, 65536(0x10000) kiW x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1

Local bus window information
Window 0  (Target FPGA 0 pre  Bus base     0xF5400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 1  (Target FPGA 0 non  Bus base     0xFAC00000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 2  (ADM-XRC-6TL-speci  Bus base     0xFAAFF000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
Window 3  (ADB3 bridge regis  Bus base     0xFAAFE000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
```

Now run the **simple** example application:

```
$ simple/simple
```

It prompts the user to enter hexadecimal values (up to 32 bits), and displays these value nibble-reversed. The nibble-reversal is performed by the target FPGA. Hitting CTRL-D exits this example. The output looks like this:

```
====================
Enter values for I/O
(use 55AA to exit)
====================
1234abcd
OUT = 0x1234abcd, IN = 0xdcba4321
55aa
OUT = 0x000055aa, IN = 0xaa550000
```

If everything works as described above, then the hardware, driver and SDK are all correctly installed and substantially working. Possible next steps are:

- Experiment with modifying and rebuilding the **simple** example application in order to become familiar with the basics of the ADMXRC3 API.
- Experiment with modifying and rebuilding the **simple** example FPGA design in order to become familiar with creating FPGA designs for Alpha Data hardware.

3 Example applications

The example applications and utilities are described in the following subsections.

DUMP	Utility for reading and writing memory access windows
FLASH	Utility for programming FPGA bitstream (.BIT) files in user-programmable Flash memory
INFO	Utility for displaying information about a reconfigurable computing device
ITEST	Example demonstrating how to consume target FPGA interrupt notifications in an application
MONITOR	Utility that displays sensor readings
SIMPLE	Example demonstrating how to read and write registers in a target FPGA
SYSMON	Utility that combines the functionality of the INFO and MONITOR utilities in a graphical user interface
VPD	Utility that allows the Vital Product Data of a reconfigurable computing device to be read or written

3.1 Building the example applications

3.1.1 Building the example applications in Windows

A Microsoft Visual Studio 2008 solution is provided, containing all of the Windows examples. This file is **%ADMXRC3_SDK%\apps\win32\apps.sln**. To build all of the examples, use the "Batch Build" command in Visual Studio.

3.1.2 Building the example applications in Linux

To build all of the example applications, excluding the **SYSMON** utility, at once, enter the following shell commands in a BASH shell:

```
$ cd $ADMXRC3_SDK/apps/linux
$ ./configure
$ make clean all
```

When compiling on 64-bit bi-architecture machine such as x86_64, two executables are built for each example application: a 64-bit native version and a 32-bit version. For example, the native version of **INFO** is named **info**, and the 32-bit version is **info32**. For machines that are not bi-architecture, only the native version is built. The **configure** script determines whether or not to build bi-architecture versions of the example applications.

The **SYSMON** utility must be built separately, because it depends upon certain packages being present in the system. For further details, refer to **Section 3.8.2, "Building SYSMON in Linux"**.</b>

3.2 DUMP utility

3.2.1 Usage

Command line

```
dump [option ...] rb window offset [n]
dump [option ...] rw window offset [n]
dump [option ...] rd window offset [n]
dump [option ...] rq window offset [n]
dump [option ...] wb window offset [n] [data ...]
dump [option ...] ww window offset [n] [data ...]
dump [option ...] wd window offset [n] [data ...]
dump [option ...] wq window offset [n] [data ...]
```

where

window	is the memory window to read or write
offset	is the offset into the window at which to begin reading or writing
n	is the number of bytes to read or write
data	is an optional data item, valid for write commands

and the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-sn <*#*>	Specifies the serial number of the card to open.
-be	Causes the data to be read or written to be treated as little-endian (default).
+be	Causes the data to be read or written to be treated as big-endian.
-hex	Causes write values to be interpreted as decimal unless prefixed by '0x' (default).
+hex	Causes write values to be interpreted as hexadecimal always.

Summary

Displays data read from a memory access window, or writes data to a memory access window.

Description

The **DUMP** utility operates in of two modes:

- Reading data from a memory access window and displaying it; for this mode, use the **rb**, **rw**, **rd** or **rq** commands.
- Writing data to a memory access window; for this mode, use the **wb**, **ww**, **wd** or **wq** commands.

In either mode, the option **+be** may be passed, before the command. This causes the **DUMP** utility to adopt big-endian byte ordering convention as opposed to little-endian (the default).

Read mode

The read command implies the radix for displaying data:

- **rb**
  Byte (8-bit) reads; data is displayed as bytes.

- **rw**
  Word (16-bit) reads; data is displayed as words.
- **rd**
  Doubleword (32-bit) reads; data is displayed as doublewords.
- **rq**
  Quadword (64-bit) reads; data is displayed as quadwords.

After the read command, a window index and an offset must be supplied, in that order. This specifies the memory access window to be read, and where in that window to begin reading data. An optional length parameter, in bytes, can also be supplied. If omitted, the length is equal to the radix implied by the read command. If present, the length parameter specifies how many bytes to read and display. The length should be an integer multiple of the width; if not, the length is rounded down.

For example, the command

```
dump rw 0 0x80000 0x60
```

produces output of the form

```
Window 0 offset 0x80000 mapped @ 0x00150000
Dump of memory at 0x00150000 + 96(0x60) bytes:
              00   02   04   06   08   0a   0c   0e
0x00150000: 000e 000f 000c b456 c567 d678 5a5a eeee ......V.g.x.ZZ..
0x00150010: eeee eeee ee22 eeee eeee eeee eeee eeee ...."..........
0x00150020: eeee eeee eeee eeee eeee eeee eeee eeee ................
0x00150030: afa7 f596 445d 8232 163f 8414 1d1e 171b ....]D2.?.......
0x00150040: c294 fa5c cd61 d464 d39d 1eed 69f8 f13d ..\.a.d......i=.
0x00150050: 5858 f489 20ff b77b ef92 a43a 6a27 e620 XX... {...:.'j .
```

Write mode

The write command implies the radix (that is, word size) to be used when performing writes:

- **wb**
  Data is written as bytes (8-bit).
- **ww**
  Data is written as words (16-bit).
- **wd**
  Data is written as doublewords (32-bit).
- **wq**
  Data is written as quadwords (64-bit).

After the write command, a window index and an offset must be supplied, in that order. This specifies the memory access window to be read, and where in that window to begin writing data. An optional length parameter, in bytes, can also be supplied. If omitted, the length is equal to the radix implied by the write command. If present, the length parameter specifies how many bytes to write. The length should be an integer multiple of the width; if not, the length is rounded down.

The program obtains the values to be written in two ways: from any additional parameters on the command line after the length parameter, and then from the standard input stream (stdin). This works as follows:

1. Any remaining command line arguments, if present after the length parameter, are interpreted as data values to be written. These values are assumed to be of the radix implied by the command, and are written to the memory window, incrementing the offset with each word written. If there are enough values passed on the command line to satisfy the byte count, the program terminates.

2. If there are insufficient data values passed on the command line, the program waits for values to be entered on the standard input stream. Values entered this way are also assumed to be of the radix implied by the command, and are written to the memory window, incrementing the offset with each word written. When the entire byte count that was specified in the length parameter has been satisfied or end-of-file is encountered, the program terminates.

An example session looks like this:

```
C>dump rd 0 0x80000 0x40
Window 0 offset 0x80000 mapped @ 0x002D0000
Dump of memory at 0x002D0000 + 80(0x40) bytes:
                  00       04       08       0c
0x002d0000: 00000000 00000000 00000000 00000000 ................
0x002d0010: 00000000 00000000 00000000 00000000 ................
0x002d0020: 00000000 00000000 00000000 00000000 ................
0x002d0030: 00000000 00000000 00000000 00000000 ................

C>dump wd 0 0x80004 0x8 0xdeadbeef
Window 0 offset 0x80004 mapped @ 0x00110004
0x80004: 0xDEADBEEF
0x80008: 0xcafeface

C>dump rd 0 0x80000 0x40
Window 0 offset 0x80000 mapped @ 0x00110000
Dump of memory at 0x00110000 + 64(0x40) bytes:
                  00       04       08       0c
0x00110000: 00000000 deadbeef cafeface 00000000 ................
0x00110010: 00000000 00000000 00000000 00000000 ................
0x00110020: 00000000 00000000 00000000 00000000 ................
0x00110030: 00000000 00000000 00000000 00000000 ................
```

Remarks

When entering data for write commands, values are expressed in decimal by default. To express data as hexadecimal, prefix it with '0x' or use the **+hex** option.

The **DUMP** utility uses store instructions for writes that are equal to the width specified on the command line, if possible. This is not possible if the CPU architecture in use does not have store instructions of the required width or if the offset specified on the command line would result in unaligned stores. In the case of an unaligned offset, writes are performed as a sequence of byte stores, because unaligned stores are illegal on some CPU architectures.

3.3 FLASH utility

3.3.1 Usage

WARNING: Incorrect use of the **+failsafe** option may impact long-term reliability of a reconfigurable computing card. Please refer to {{link :section:Failsafe bitstream mechanism:$r}} below for an explanation of the **+failsafe** option and how it may be used.

Command line

```
flash [option ...] chkblank target-index
flash [option ...] erase    target-index
flash [option ...] program  target-index filename
flash [option ...] verify   target-index filename
```

where

target-index	is the index of a target FPGA
filename	is the name of a .BIT file (program or verify commands only)

and the following options are accepted:

-index *<index>*	Specifies the index of the card to open (default 0).
-sn *<#>*	Specifies the serial number of the card to open.
-failsafe	Causes the normal image to be erased / programmed / verified (default).
+failsafe	Causes the failsafe image to be erased / programmed / verified; see **Failsafe bitstream mechanism** below.
-force	Causes a mismatch between the target FPGA device and the .BIT file device to result in an error (default).
+force	Causes a mismatch between the target FPGA device and the .BIT file device to be ignored.

Summary

Blank-checks, erases, programs or verifies a target FPGA bitstream image in the user-programmable Flash memory of a device.

Description

The **FLASH** utility has four commands:

- chkblank *<target-index>*
  Verifies that an image is blank, i.e. all bytes are 0xFF.
- erase *<target-index>*
  Erases an image so that it becomes become, i.e. all bytes are 0xFF.
- program *<target-index> <filename>*
  Programs the specified bitstream (.BIT) file into an image so that the target FPGA is configured from the image at power-on or reset.
- verify *<target-index> <filename>*
  Verifies that an image contains the specified bitstream (.BIT) file.

chkblank command

The **chkblank** command verifies that a target FPGA image is blank, i.e. all bytes are 0xFF, but does not modify the Flash memory bank. Following the command, an index of a target FPGA in the device must be specified. The index of the target FPGA is normally zero but may be nonzero in in models with multiple target FPGAs.

For example, to blank-check the default image for target FPGA 0:

```
flash program 0 /path/to/my_design.bit
```

erase command

The **erase** command erases a target FPGA image so that it becomes blank, i.e. all bytes are 0xFF. It automatically performs a blank-check after erasing. Following the command, an index of a target FPGA in the device must be specified. The index of the target FPGA is normally zero but may be nonzero in in models with multiple target FPGAs.

For example, to erase the default image for target FPGA 0:

```
flash erase 0
```

program command

The **program** command programs a target FPGA image with the data in the specified bitstream (.BIT) file. Following the command, an index of a target FPGA in the device and the name of a bitstream (.BIT) filename must be specified. The index of the target FPGA is normally zero but may be nonzero in in models with multiple target FPGAs.

If the device in the .BIT file does not match the target FPGA, this command fails with an error and does not program the target FPGA image, unless the **+force** option is passed. Verification is automatically performed after programming.

For example, to program the default image for target FPGA 0 with a bitstream file called **my_design.bit**:

```
flash program 0 /path/to/my_design.bit
```

verify command

The **verify** command verifies that a target FPGA image contains the data in the specified bitstream (.BIT) file, but does not modify the Flash memory bank. Following the command, an index of a target FPGA in the device and the name of a bitstream (.BIT) filename must be specified. The index of the target FPGA is normally zero but may be nonzero in in models with multiple target FPGAs.

If the device in the .BIT file does not match the target FPGA, this command fails with an error unless the **+force** option is passed. If discrepancies between the target FPGA image and the data in the .BIT file are found, they are displayed (up to a certain number of erroneous bytes), followed by a failure message.

For example, to verify that the default image for target FPGA 0 contains the data in a bitstream file called **my_design.bit**:

```
flash verify 0 /path/to/my_design.bit
```

Failsafe bitstream mechanism

Due to errata in certain Xilinx™ FPGA families, the following Gen 3 models have a "failsafe bitstream" mechanism:

- ADM-XRC-6TL
- ADM-XRC-6T1

In the above models, each target FPGA has two images: a default image, and a failsafe image. Alpha Data factory-programs a known-good "null bitstream" into the failsafe image. When power is applied to a card, the firmware on the card first looks for a valid bitstream in the default image. If no bitstream is found, the firmware uses the null bitstream in the failsafe image to configure the target FPGA. In this way, the firmware ensures that the target FPGA is always configured with something when it is powered-on.

Because the purpose of the failsafe image is to protect the target FPGA from sub-micron effects that would otherwise degrade the performance of the target FPGA over time, Alpha Data recommends that the failsafe image should never be erased. If overwritten, a customer must ensure that the bitstream is valid, known-good and satisfies the requirements for protecting the target FPGA from sub-micron effects.

Xilinx™ answer record 35055 elaborates on protecting Virtex-6 GTX transceivers from performance degradation over time.

3.4 INFO utility

3.4.1 Usage

Command line

```
info [option ...]
```

where the following options are accepted:

-index *<index>*	Specifies the index of the card to open (default 0).
-sn <#>	Specifies the serial number of the card to open.
-flash	Causes Flash bank information not to be shown (default).
+flash	Causes Flash bank information to be shown.
-io	Causes I/O module information not to be shown (default).
+io	Causes I/O module information to be shown.
-sensor	Causes sensor information not to be shown (default).
+sensor	Causes sensor information to be shown.

Summary

Displays information about a reconfigurable computing device.

Description

The **INFO** utility demonstrates the use of most of the informational functions in the ADMXRC3 API. It uses **ADMXRC3_OpenEx** to open a device in passive mode, meaning that an unprivileged user can successfully run it. The output consists of several sections, the first of which is obtained using **ADMXRC3_GetVersionInfo**:

```
API information
API library version          1.1.0
Driver version               1.1.0
```

The second section shows information obtained using **ADMXRC3_GetCardInfoEx**, and shows the information in the **ADMXRC3_CARD_INFOEX** structure:

```
Card information
Model                        ADM-XRC-6TL
Serial number                101(0x65)
Number of programmable clocks 1
Number of DMA channels       1
Number of target FPGAs       1
Number of local bus windows  4
Number of sensors            10
Number of I/O module sites   1
Number of local bus windows  4
Number of memory banks       4
Bank presence bitmap         0xF
```

The third section uses the **NumTargetFpga** member of the **ADMXRC3_CARD_INFOEX** structure and **ADMXRC3_GetFpgaInfo** to enumerate the target FPGAs in the device:

```
Target FPGA information
FPGA 0                       xc6vlx240tff1759
```

The fourth section uses the **NumMemoryBank** member of the **ADMXRC3_CARD_INFOEX** structure and **ADMXRC3_GetBankInfo** to enumerate the memory banks (non-Flash) in the device:

```
Memory bank information
```

```
Bank 0                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 1                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 2                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
Bank 3                        SDRAM, DDR3, 65536 kiWord x 32+0 bits
                              303.0 MHz - 533.3 MHz
                              Connectivity mask 0x1
```

The fourth section uses the **NumWindow** member of the **ADMXRC3_CARD_INFOEX** structure and **ADMXRC3_GetWindowInfo** to enumerate the memory access windows in the device:

```
Local bus window information
Window 0  (Target FPGA 0 pre Bus base     0xF5400000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 1  (Target FPGA 0 non Bus base     0xFAC00000 size 0x400000
                              Local base   0x0 size 0x400000
                              Virtual size 0x400000
Window 2  (ADM-XRC-6TL-speci Bus base     0xFAAFF000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
Window 3  (ADB3 bridge regis Bus base     0xFAAFE000 size 0x1000
                              Local base   0x0 size 0x0
                              Virtual size 0x1000
```

The next section appears if the **+flash** option is passed on the command line. It uses the **NumFlashBank** member of the **ADMXRC3_CARD_INFOEX** structure and **ADMXRC3_GetFlashInfo** to enumerate the Flash memory banks in the device:

```
Flash bank information
Bank 0                        Intel 28F256P30, 65536(0x10000) kiB
                              Useable area 0x1200000-0x3FFFFFF
```

The next section appears if the **+io** option is passed on the command line. It uses the **NumModuleSite** member of the **ADMXRC3_CARD_INFOEX** structure and **ADMXRC3_GetModuleInfo** to enumerate the I/O module sites in the device and show what is fitted, if anything:

```
I/O module information
Module 0                      Not present
```

The final section appears if the **+sensor** option is passed on the command line. It uses the **NumSensor** member of the **ADMXRC3_CARD_INFOEX** structure and **ADMXRC3_GetSensorInfo** to enumerate the sensors in the device:

```
Sensor information
Sensor 0                      1V supply rail
                              V, double, exponent 0, error 0.0
Sensor 1                      1.5V supply rail
                              V, double, exponent 0, error 0.0
Sensor 2                      1.8V supply rail
                              V, double, exponent 0, error 0.0
Sensor 3                      2.5V supply rail
                              V, double, exponent 0, error 0.1
Sensor 4                      3.3V supply rail
                              V, double, exponent 0, error 0.1
Sensor 5                      5V supply rail
                              V, double, exponent 0, error 0.1
Sensor 6                      XMC variable power rail
                              V, double, exponent 0, error 0.2
Sensor 7                      XRM I/O voltage
                              V, double, exponent 0, error 0.1
Sensor 8                      LM87 internal temperature
```

```
                              deg. C, double, exponent 0, error 3.0
    Sensor 9                  Target FPGA temperature
                              deg. C, double, exponent 0, error 4.0
```

3.5 ITEST example

3.5.1 Usage

Command line

```
itest [option ...]
```

where the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-sn <#>	Specifies the serial number of the card to open.

Summary

Demonstrates consumption of FPGA interrupt notifications.

Description

This example demonstrates how to consume FPGA interrupt notifications in an application. It uses the interrupt test block of the UBER example FPGA design, described in **Section 4.5.1.3.5, "Interrupt Test Block"** as a means of generating FPGA interrupt notifications, and starts a thread whose purpose is to wait for and acknowledge interrupts from the target FPGA.

When ITEST is started, the main thread first configures target FPGA 0 with the **Section 4.5, "Uber Example FPGA Design"**. The main thread then launches an interrupt thread that waits for notifications, in a loop. The main thread then proceeds to wait for input, also in a loop. At this point, the user may press RETURN to generate an interrupt, or enter 'q' to terminate the program. On termination, the program displays the number of FPGA interrupt notifications that the interrupt thread consumed during execution.

A sample session looks like this:

```
Enter 'q' to quit, or anything else to generate an interrupt:
Interrupt thread started

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:

Enter 'q' to quit, or anything else to generate an interrupt:
q
Generated 5 interrupts
Interrupt thread saw 5 interrupt(s)
```

The blank lines in the above session are simply empty lines where the user has pressed return. As can be seen, each of the 5 interrupts generated results in the interrupt thread consuming a notification.

Remarks

As noted in the ADMXRC3 API Specification (see functions **ADMXRC3_RegisterWin32Event**, **ADMXRC3_RegisterVxwSem** and **ADMXRC3_StartNotificationWait**), the ADMXRC3 API does not queue each type of notification. Therefore, this example works as expected as long as the frequency of target FPGA interrupt notifications is not too fast for the interrupt thread. Since the rate of generation of notifications in this example is limited the user's keyboard input rate, the interrupt thread should be able to keep up (as long as the machine is not heavily loaded with other processes). Nevertheless, it is important to note that in this simple example, there is no mechanism for throttling the rate of notifications so that notifications cannot be lost. In a real application, the preferred design approaches are:

1. Architect the FPGA design and host application so that they tolerate *out-of-date* notifications being missed. For example, if the target FPGA generates an interrupt when data arrives via an I/O interface, it does not matter if the host application does not succeed in consuming every target FPGA interrupt notification, because the notifications before the latest one are considered out-of-date. When the host application handles a notification, it reads a register in the target FPGA to determine the amount of new data rather than using the number of notifications consumed. What matters is that regardless of how many times the target FPGA generates an interrupt, the host application is guaranteed to eventually wake up and check for new data.
2. Use a fully handshaked system, where the host application must positively acknowledge a target FPGA interrupt before the target FPGA generates a new interrupt.

In fact, the above two approaches are best used together, because minimizing the number of FPGA interrupts minimizes unnecessary context switches in the operating system.

3.6 MONITOR utility

3.6.1 Usage

Command line

```
monitor [option ...]
```

where the following options are accepted:

-index *<index>*	Specifies the index of the card to open (default 0).
-sn *<#>*	Specifies the serial number of the card to open.
-period *<delay>*	Specifies the update period, in seconds.
-repeat *<n>*	Specifies the number of repetitions (default 0) A value of zero means "repeat for ever".

Summary

Displays readings from all sensors.

Description

The **MONITOR** utility repeatedly displays sensor readings in the command shell at the interval specified by the **-period** option. The number of updates to perform before terminating can be specified on the command line using the **-repeat** option, but by default, the program runs until interrupted with CTRL-C.

It makes use of the **ADMXRC3_GetSensorInfo** and **ADMXRC3_ReadSensor** functions from the ADMXRC3 API, and because it opens a device in passive mode using **ADMXRC3_OpenEx**, it can run alongside other reconfigurable computing applications without disturbing them.

The output looks like this:

```
Model:                                    257 (0x101) => ADM-XRC-6TL
Serial number:                            101 (0x65)
Number of sensors:                        10
  Sensor  0               1V supply rail: 0.987000 V
  Sensor  1             1.5V supply rail: 1.509186 V
  Sensor  2             1.8V supply rail: 1.803192 V
  Sensor  3             2.5V supply rail: 2.508896 V
  Sensor  4             3.3V supply rail: 3.268082 V
  Sensor  5               5V supply rail: 5.017990 V
  Sensor  6      XMC variable power rail: 12.000000 V
  Sensor  7              XRM I/O voltage: 2.495712 V
  Sensor  8    LM87 internal temperature: 49.000000 deg C
  Sensor  9      Target FPGA temperature: 57.000000 deg C
```

3.7 SIMPLE example

3.7.1 Usage

Command line

```
simple [option ...]
```

where the following options are accepted:

-index *<index>*	Specifies the index of the card to open (default 0).
-sn <#>	Specifies the serial number of the card to open.
-uber	Uses SIMPLE FPGA design (default).
+uber	Uses UBER FPGA design.

Summary

Demonstrates access to target FPGA registers.

Description

The SIMPLE example application demonstrates accessing FPGA registers in its simplest form. It first configures target FPGA 0 with the **Section 4.4, "Simple Example FPGA Design"** and then waits for input from the user. The user enters a hexadecimal value (up to 32 bits in length), and the program writes each one to a register in the target FPGA. The target FPGA nibble-reverses the value (i.e. swaps bits 31:28 with 3:0, 27:24 with 7:4 etc.) and the program reads back the nibble-reversed value and displays it. The program terminates when the value 55AA (hex) is entered.

A sample session looks like this:

```
====================
Enter values for I/O
 (use 55AA to exit)
====================
12345678
OUT = 0x12345678, IN = 0x87654321
deadbeef
OUT = 0xdeadbeef, IN = 0xfeebdaed
cafeface
OUT = 0xcafeface, IN = 0xecafefac
55aa
OUT = 0x000055aa, IN = 0xaa550000
```

3.8 SYSMON utility

3.8.1 Usage

Command line

```
sysmon
```

Summary

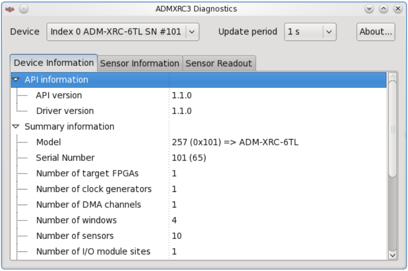
Utility presenting device information and hardware sensors in a graphical user interface.

Description

The **SYSMON** utility combines the information shown by the INFO and MONITOR utilities with a graphical user interface. Its main function is graphical display of hardware sensor data, and it can be minimized to the notification area of the desktop (the "System Tray" in Windows) in order to run unobtrusively.

It makes use of the **ADMXRC3_GetSensorInfo** and **ADMXRC3_ReadSensor** functions from the ADMXRC3 API, and because it opens a device in passive mode using **ADMXRC3_OpenEx**, it can run alongside other reconfigurable computing applications without disturbing them.

The user interface of the Linux version of SYSMON is as follows upon starting the utility:

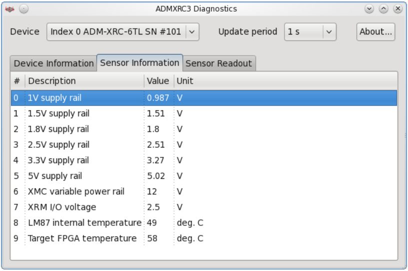


Device: Index 0 ADM-XRC-6TL SN #101
Update period: 1 s

API information	
API version	1.1.0
Driver version	1.1.0
Summary information	
Model	257 (0x101) => ADM-XRC-6TL
Serial Number	101 (65)
Number of target FPGAs	1
Number of clock generators	1
Number of DMA channels	1
Number of windows	4
Number of sensors	10
Number of I/O module sites	1

Figure 2: SYSMON user interface - device information

The Windows version of SYSMON offers equivalent functionality, but uses a different GUI technology to that of the Linux version. The second tab shows sensor readings in tabular form:



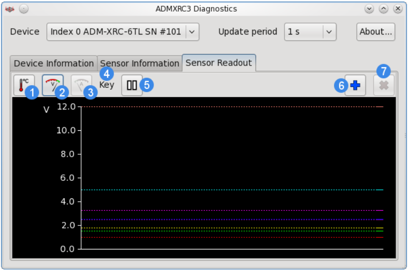
Device: Index 0 ADM-XRC-6TL SN #101

Update period: 1 s

#	Description	Value	Unit
0	1V supply rail	0.987	V
1	1.5V supply rail	1.51	V
2	1.8V supply rail	1.8	V
3	2.5V supply rail	2.51	V
4	3.3V supply rail	3.27	V
5	5V supply rail	5.02	V
6	XMC variable power rail	12	V
7	XRM I/O voltage	2.5	V
8	LM87 internal temperature	49	deg. C
9	Target FPGA temperature	58	deg. C

Figure 3: SYSMON user interface - sensor readings

The third tab displays sensor readings in graphical form:



Device: Index 0 ADM-XRC-6TL SN #101

Update period: 1 s

Figure 4: SYSMON user interface - sensor display

Initially, the 'scope is empty and displays no sensors. The above figure shows the effect of clicking the voltage button, labelled 2 in the above figure. The user interface elements of the 'scope toolbar are as follows:

1. The temperature button sets the 'scope to display all temperature sensors in the device. Once some sensors are displayed, updates begin.
2. The voltage button sets the 'scope to display all voltage sensors in the device. Once some sensors are displayed, updates begin.
3. The current button sets the 'scope to display all current sensors in the device. Once some sensors are displayed, updates begin.
4. The key can be moused-over to show which sensor corresponds to which colored trace.
5. The pause / resume button can be used to pause and resume update of the 'scope.
6. Item 6 is a button that adds another 'scope when clicked, to a maximum of 4, so that various types of sensor can be viewed at the same time.
7. Item 7 is a button that destroys a 'scope when clicked. If there is only one 'scope, the button is disabled.

3.8.2 Building SYSMON in Linux

The Linux version of the **SYSMON** utility uses **GTKMM-2.4**. This package is present in recent Linux distributions such as Fedora Core 13, but may not be present in all Linux distributions. For this reason, **SYSMON** is built separately from the other example applications. A non-exhaustive list of the packages that are required to build **SYSMON** is as follows:

gtkmm24-devel
libsigc++20-devel
pangomm-devel
cairomm-devel
glibmm24-devel
pkgconfig

To run **SYSMON**, the corresponding runtime packages are required:

gtkmm24
libsigc++20
pangomm
cairomm
glibmm24

To build the "Release" configuration of **SYSMON**, enter the following commands in a BASH shell:

```
$ cd $ADMXRC3_SDK/apps/linux
$ ./configure
$ cd sysmon
$ make CONFIG=Release clean all
```

The executable's path is then **$ADMXRC3_SDK/apps/linux/sysmon/bin/Release/sysmon**.

3.9 VPD utility

3.9.1 Usage

Command line

```
vpd [option ...] fb address n [data]
vpd [option ...] fw address n [data]
vpd [option ...] fd address n [data]
vpd [option ...] fq address n [data]
vpd [option ...] fs address n [string]
vpd [option ...] rb address [n]
vpd [option ...] rw address [n]
vpd [option ...] rd address [n]
vpd [option ...] rq address [n]
vpd [option ...] wb address [n] [data ...]
vpd [option ...] ww address [n] [data ...]
vpd [option ...] wd address [n] [data ...]
vpd [option ...] wq address [n] [data ...]
vpd [option ...] ws address [n] [string ...]
```

where

address	is the address in VPD memory at which to begin reading or writing
n	is the number of bytes to read or write
data	is an numeric data item, valid for fill and write commands
string	is an string data item, valid for fill and write commands

and the following options are accepted:

-index <*index*>	Specifies the index of the card to open (default 0).
-sn <*#*>	Specifies the serial number of the card to open.
-hex	Causes numeric data values to be interpreted as decimal unless prefixed by '0x' (default).
+hex	Causes numeric data values to be interpreted as hexadecimal always.

Summary

Displays data read from VPD memory, or writes data to VPD memory.

Description

The **VPD** utility operates in one of three modes:

- Filling a region of VPD memory with a value or string; for this mode, use the **fb**, **fw**, **fd**, **fq** or **fs** commands.
- Reading data from VPD memory and displaying it; for this mode, use the **rb**, **rw**, **rd** or **rq** commands.
- Writing numeric or string data to a region of VPD memory; for this mode, use the **wb**, **ww**, **wd**, **wq** or **ws** commands.

Fill mode

When filling a region of VPD memory with data, the fill command specifies whether the data is numeric or string data. In the case of numeric data, the command also implies the radix (i.e. word size) of the data. The available fill commands are:

- **fb**
  Fill value is a byte (8-bit).
- **fw**
  Fill value is a word (16-bit).
- **fd**
  Fill value is a doubleword (32-bit).
- **fq**
  Fill value is a quadword (64-bit).
- **fs**
  Fill value is an ASCII string (8-bit characters).

The next 3 arguments after the fill command must be:

(a) *address* - the byte address within VPD memory at which to begin filling
(b) *n* - byte count; the number of bytes of VPD memory to fill
(c) *data* or *string* - the numeric or string value to place in the specified region of VPD memory

If the command is **fs** and the string value is shorter than the byte count *n*, the string is repeated until the byte count is satisfied. If the string is longer than the byte count *n*, only the first *n* characters are used. If a string contains spaces, it must be quoted on the command line so that it is not interpreted by the shell as two or more separate arguments.

For the numeric fill commands **fb**, **fw**, **fd** and **fq**, the numeric value is repeated until the byte count is satisfied.

Read mode

The read command implies the radix (i.e. word size) used for displaying the data:

- **rb**
  Byte (8-bit) reads; data is displayed as bytes.
- **rw**
  Word (16-bit) reads; data is displayed as words.
- **rd**
  Doubleword (32-bit) reads; data is displayed as doublewords.
- **rq**
  Quadword (64-bit) reads; data is displayed as quadwords.

After the read command, an address must be supplied, which specifies where in VPD memory to begin reading. An optional length parameter, in bytes, can also be supplied. If omitted, the length is equal to the radix implied by the read command. If present, the length parameter specifies how many bytes to read and display. The length should be an integer multiple of the width; if not, the length is rounded down.

Write mode

The write command specifies whether the data is numeric or string data. In the case of numeric data, the command also implies the radix (i.e. word size) of the data. The available write commands are:

- **wb**
  Data is written as bytes (8-bit).
- **ww**
  Data is written as words (16-bit).
- **wd**
  Data is written as doublewords (32-bit).
- **wq**
  Data is written as quadwords (64-bit).

- **ws**
  Data is supplied as one or more ASCII trings (8-bit characters).

After the write command, an address must be supplied, which specifies where in VPD memory to begin writing data. An optional length parameter, in bytes, can also be supplied. If omitted, the length is equal to the radix implied by the write command. If present, the length parameter specifies how many bytes to write. The length should be an integer multiple of the width; if not, the length is rounded down.

The program obtains the values to be written in two ways: from any additional parameters on the command line after the length parameter, and then from the standard input stream (stdin). This works as follows:

1. Any remaining command line arguments, if present after the length parameter, are interpreted as data values to be written. Numeric values are assumed to be of the radix implied by the command parameter. As each value it written to VPD memory, the address is incremented. If there are enough values passed on the command line to satisfy the byte count, the program terminates.
2. If there are insufficient data values passed on the command line, the program waits for values to be entered on the standard input stream. Numeric values entered this way are also assumed to be of the radix implied by the command. As each value it written to VPD memory, the address is incremented. When the entire byte count that was specified in the length parameter has been satisfied or end-of-file is encountered, the program terminates.

Example session

The following session was captured under Linux using an ADM-XRC-6TL. The base address 0x100000 is used because that is the VPD-space address of the user-definable area of VPD memory in the ADM-XRC-6TL.

```
$ ./vpd rb 0x100000 0x60 0xff
Dump of VPD at 0x100000 + 96(0x60) bytes:
            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0x00100000: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100010: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100020: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100030: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100040: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100050: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
$ ./vpd fs 0x100008 20 'hello world!'
$ ./vpd wd 0x100020 12
0x00100020: 0xdeadbeef
0x00100024: 0xcafeface
0x00100028: 0x12345678
$ ./vpd fw 0x100031 10 0xa55a
$ ./vpd rb 0x100000 0x60
Dump of VPD at 0x100000 + 96(0x60) bytes:
            00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
0x00100000: ff ff ff ff ff ff ff ff 68 65 6c 6c 6f 20 77 6f ........hello wo
0x00100010: 72 6c 64 21 68 65 6c 6c 6f 20 77 6f ff ff ff ff rld!hello wo....
0x00100020: ef be ad de ce fa fe ca 78 56 34 12 ff ff ff ff ........xV4.....
0x00100030: ff 5a a5 5a a5 5a a5 5a a5 5a a5 ff ff ff ff ff .Z.Z.Z.Z.Z......
0x00100040: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
0x00100050: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................
```

Remarks

When entering data for fill or write commands, values are expressed in decimal by default. To express data as hexadecimal, prefix it with '0x' or use the **+hex** option.

In the current version of the **VPD** utility, data is always read from and written to VPD memory in little-endian byte order.

4 Example HDL FPGA Designs

4.1 Introduction

A number of example FPGA designs are included with the SDK. The purpose of these is to demonstrate functionality available on the Virtex 6 based ADM-XRC series of cards and also to serve as customisable starting points for user-developed designs. A testbench and simulation/build scripts are also included with each example design.

The example applications use these example designs to demonstrate how software running on the host CPU can interact with an FPGA design.

The table below lists the example FPGA designs and their related applications:

FPGA Design	Host Application	Purpose
simple	simple	Demonstrates implementation of host-accessible registers. Uses a signal naming convention consistent with the Local Bus in earlier generations of the ADM-XRC.
uber	simple	Demonstrates implementation of host-accessible registers.
uber	itest	Demonstrates implementation of FPGA interrupts.

Table 1: FPGA Designs/Host applications

Example designs are located in the **%ADMXRC3_SDK%\hdl\vhdl\examples** directory.

4.2 Design Simulation Using Modelsim

A testbench design and macro files compatible with Modelsim are provided for simulation of each example FPGA design. For details specific to each example design, refer to its **Design Simulation** section.

4.3 Bitstream Build Using ISE

Bitstreams for all supported combinations of example FPGA design, board, and device are supplied pre-built in the **%ADMXRC3_SDK%\bit** directory of the SDK. This directory is the equivalent of the **%ADMXRC3_SDK%\bin** directory for the example applications. The sources files required to re-build all bitstreams are supplied in the **%ADMXRC3_SDK%\hdl** directory. Bitstream build using the Windows environment requires the use of the Visual Studio nmake command. Bitstream build using the Linux environment requires the use of the GNU Make command.

4.3.1 Building All Example Bitstreams

A makefile is provided for building all bitstreams for all example FPGA designs. It is located in the **%ADMXRC3_SDK%\hdl\vhdl\examples** directory. As many bitstream files will be generated, it may take from minutes to hours to run to completion. To perform the build using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples
nmake clean all
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples
make clean all
```

To perform a build and install the resulting bitstream files in the **%ADMXRC3_SDK%\bit** directory using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples
nmake clean install
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples
make clean install
```

4.3.2 Building Specific Example/Board/Device Bitstreams

For each example FPGA design, a makefile is provided for building all its bitstreams, or a specific board/device bitstream. For details specific to each example design, refer to its **Bitstream Build** section.

4.4 Simple Example FPGA Design

4.4.1 Design Description

The Simple example FPGA design demonstrates direct slave register access on the Virtex 6 series of ADM-XRC boards. The design includes the following functional areas:

- Clock Generation

- Internal clock generation
- External clock buffering (non-MGT sourced) and extraction (MGT sourced)

- MPTL Interface (mptl_if_target_wrap)

- OCP Direct Slave Channel

- Simple test using host-accessible registers

The Simple example FPGA design top level **simple_l.vhd** is located in **%ADMXRC3_SDK%\hdl\vhdl\examples\simple\common**. It consists of the following blocks:

- MPTL interface block (mptl_if_target_wrap)
- OCP Direct Slave interface block (adb3_ocp_simple_bus_if)

A top level block diagram of the Simple example design is shown in **Figure 5, "Simple Design Testbench And Top Level Block Diagram"**.

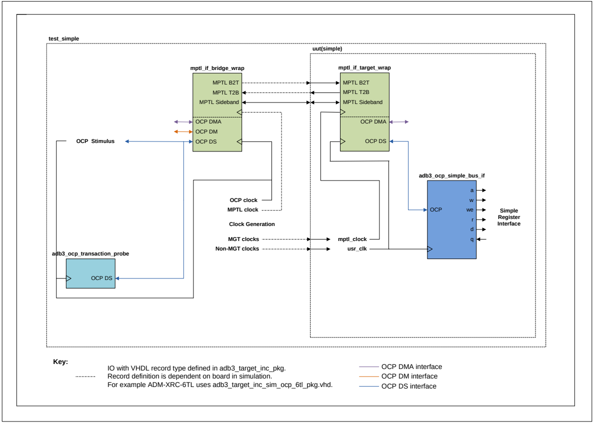


Figure 5: Simple Design Testbench And Top Level Block Diagram

4.4.1.1 Clock Generation

This function includes the following functional areas:

- Internal clock generation
- External clock buffering (non-MGT sourced)
- External clock extraction (MGT sourced)
- MPTL interface clock generation

4.4.1.1.1 Internal Clock Generation

A user clock **usr_clk** is generated from a buffered version of the ref_clk input.

4.4.1.1.2 External Clock Buffering (Non-MGT Sourced)

Non-MGT clock inputs are buffered. Clock support is dependent on board selected. Refer to **Figure 10, "Uber Design Clock Buffering/Extraction"**.

4.4.1.1.3 External Clock Extraction (MGT Sourced)

MGT clock inputs are converted from double-ended to single-ended and then buffered. The buffered clocks are connected to the clk_vec signal. The connection order is defined in the uber_pkg.vhd file. Clock support is dependent on board selected.

4.4.1.1.4 MPTL interface clock generation

The MPTL interface block requires an mptl_clk clock input. This is generated from an FPGA MGT clock input. The **mptl_clk** signal may be single or double ended depending on the board in use. Its type **mptl_clk_t**is defined in the board specific package **adb3_target_inc_pkg** which is located in the board directory in **%ADMXRC3_SDK%\fpga\common\adb3_target**.

During simulation, the **mptl_clk_t** record needs to contain both single and double ended clock record elements. Only the record elements appropriate to the board being simulated are driven.

During synthesis, the **mptl_clk_t** record need only contain the clock record elements specific to the board being built.

Two functions: **sgl_to_mptl_clk_t** and **dbl_to_mptl_clk_t** are provided in the board specific package **adb3_target_inc_pkg** to convert from the MGT clock input type to the mptl_clk_t type.

4.4.1.2 MPTL Interface

This function is implemented using the MPTL library component **mptl_if_target_wrap**. Refer to Section 5 for a functional description.

4.4.1.3 OCP Direct Slave Channel

The OCP Direct Slave Channel function consists of an OCP to parallel interface block, and a register section. The MPTL interface OCP Direct Slave channel connects to the OCP to parallel interface block. This block is implemented using the ADB3 OCP library component **adb3_ocp_simple_bus_if**. Refer to Section 5 for a functional description.

4.4.1.3.1 Simple Test Registers

4.4.1.3.1.1 Description

The OCP to parallel interface block connects to the Simple test registers. Write accesses are controlled by the write enable bus and/or the write signal. Read accesses are controlled by the read signal. The address bus is used to select the register to be accessed and data is transferred on the data busses.

4.4.1.3.1.2 Register Interface

The Simple FPGA design implements registers in the Direct Slave OCP address space as follows:

Name	Type	Address
DATA	RW	0x00000000

Table 2: Simple Design Simple Test Block Address Map

Bits	Mnemonic	Type	Function
31:0	DATA	RW	Indicates the nibble-reversed version of the last data written.

Table 3: Simple Design Simple Test Block DATA Register

4.4.2 Board Support

The Simple FPGA design is compatible with all Virtex 6 based boards.

4.4.3 Source Location

The Simple FPGA design is located in **%ADMXRC3_SDK%\fpga\examples\simple**. Source files common to all boards and are located in the **\common** directory. These include the design and testbench top levels.

For a complete list of the source files used during simulation, refer to the appropriate Modelsim macro file located in the board design directory, for example: **\admxrc6tl\simple-admxrc6tl.do** for the ADM-XRC-6TL.

For a complete list of the source files used during synthesis, refer to the appropriate XST project file located in the board design directory, for example: **\admxrc6t1\simple-admxrc6t1.prj** for the ADM-XRC-6T1.

4.4.4 Testbench Decription

The Simple example FPGA design testbench **test_simple.vhd** is located in **%ADMXRC3_SDK%\hdl\vhdl\examples\simple\common**. Refer to **Figure 5, "Simple Design Testbench And Top Level Block Diagram"**.

The design testbench consists of the following functional areas:

- Clock generation
- Test Direct Slave Interface
- Bridge MPTL interface (mptl_if_bridge_wrap)
- OCP test probe (adb3_ocp_transaction_probe)

4.4.4.1 Clock Generation

4.4.4.1.1 Simple Example Design Clocks

- The Simple example design **clks_non_mgt** input is dependent on the board selected. It is connected to appropriate clocks generated in the testbench. Refer to the testbench file for connection information for each board.
- The Simple example design **clks_mgt** input is dependent on the board selected. It is connected to appropriate clocks generated in the testbench. Refer to the testbench file for connection information for each board.

4.4.4.1.2 Testbench Clocks

The Bridge MPTL Interface **mptl_if_bridge_wrap** input **ocp_clk** connection is dependent on the type of simulation selected.

- During OCP-OCP simulation, it must be driven by the same clock as the Simple example design MPTL Interface **mptl_if_target_wrap** input **ocp_clk**. This signal is transferred to the testbench using the **mptl_t2b.target_ocp_clk** record element.

The Bridge MPTL Interface **mptl_if_bridge_wrap** input **mptl_clk** connection is dependent on the type of board selected. Refer to the testbench file for connection information for each board.

4.4.4.2 Test Direct Slave Interface

This function connects to the **mptl_if_bridge_wrap** OCP direct slave interface and contains the following sections:

4.4.4.2.1 Simple Test

This section communicates with the Simple Test block registers as follows:

```
Write (32-bit), set DATA = 0x"cafeface"
Read  (32-bit), exp DATA = 0x"cafeface"
```

Section complete and pass/fail indications are returned using the **simple_complete** and **simple_passed** signals respectively.

4.4.4.2.2 Bridge MPTL interface

This function is implemented using the MPTL library component **mptl_if_bridge_wrap**. Refer to Section 5 for a functional description.

The **mptl_if_bridge_wrap** component output **bridge_gtp_online_n** is combined with the Simple example design output **target_gtp_online_n** to produce the **mptl_online_long** signal. This indicates that the MPTL interface is active and stable. This signal is monitored and will terminate the simulation if it goes inactive.

4.4.4.2.3 OCP test probes

This function monitors the direct slave OCP interface for transaction errors using the ADB3 Probe library component **adb3_ocp_transaction_probe_sim**. Refer to Section 5 for a functional description.

4.4.5 Design Simulation

Modelsim macro files are located in **%ADMXRC3_SDK%\fpga\examples\simple** in each of the board design directories. For example **\admxrc6tl\simple-admxrc6tl.do** for the ADM-XRC-6TL.

Modelsim simulation is initiated using the **vsim** command with the appropriate macro file. For example, to perform a modelsim simulation using windows and the ADM-XRC-6TL, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
vsim -do "simple-admxrc6tl.do"
```

Expected simulation results are shown below.

4.4.5.1 Initialisation Results

Modelsim output during initialisation of simulation will be similar to the following example:

```
# ** Note: Board Type : adm_xrc_6tl
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
# ** Note: Target Use : sim_ocp
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
# ** Note: Waiting for MPTL online....
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
```

4.4.5.2 Direct Slave Test Results

Modelsim output during simulation will be similar to the following example:

```
# ** Note: Wrote simple WDATA 4 bytes 0xCAFEFACE with enable 0b1111 to byte address 0
#    Time: 1620 ns  Iteration: 6  Instance: /test_simple
# ** Note: Read simple RDATA 4 bytes 0xECAFEFAC from byte address 0
#    Time: 1687500 ps  Iteration: 6  Instance: /test_simple
# ** Note: Test Simple completed: PASSED.
#    Time: 1687500 ps  Iteration: 6  Instance: /test_simple
```

4.4.5.3 Completion Results

Modelsim output on successful completion of simulation will be similar to the following example:

```
# ** Failure: Test of design SIMPLE completed: PASSED.
#    Time: 1687500 ps  Iteration: 8  Process: /test_simple/test_results_p File: ../common/test_simple.vhd
# Break in Process test_results_p at ../common/test_simple.vhd line 244
# Simulation Breakpoint: Break in Process test_results_p at ../common/test_simple.vhd line 244
# MACRO ./simple-admxrc6tl.do PAUSED at line 34
```

4.4.6 Bitstream Build

A makefile is provided for building all bitstreams, or a specific board/device bitstream, for the Simple FPGA example design. It is located in the **%ADMXRC3_SDK%\hdl\vhdl\examples\simple** directory. In order to use these re-built bitstream with the example applications, they must be copied to the **%ADMXRC3_SDK%\bit\simple** directory. This can be performed automatically using the **install** makefile option. A "clean up" of the files produced by the build process can be performed using the **clean** makefile option. Examples are as follows:

To perform a build of all Simple design bitstreams using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
```

```
nmake clean all
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
make clean all
```

To perform a build and install the resulting bitstreams using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
nmake clean install
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
make clean install
```

To perform a build for an ADM-XRC-6TL board fitted with an 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
nmake bit_admxrc6tl_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
make bit_admxrc6tl_6vlx240t
```

To perform a build and install for an ADM-XRC-6TL board fitted with an 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
nmake inst_admxrc6tl_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
make inst_admxrc6tl_6vlx240t
```

To perform a clean for an ADM-XRC-6TL board fitted with an 6VLX240T device using Windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\simple
nmake clean_admxrc6tl_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/simple
nmake clean_admxrc6tl_6vlx240t
```

The full path and filename of bitstreams built using Windows will be:

```
%ADMXRC3_SDK%\hdl\vhdl\examples\simple\output\<design>-<board>-<device>.bit
```

The full path and filename of bitstreams built using Linux will be:

```
$ADMXRC3_SDK/hdl/vhdl/examples/simple/output/<design>-<board>-<device>.bit
```

4.4.7 ISE Constraint Files

Constraint files for building Simple design bitstream files using ISE are provided. These files are located in **%ADMXRC3_SDK%\fpga\examples\simple** in each of the board design directories, for example **\admxrc6tl\simple-admxrc6tl.ucf** for the ADM-XRC-6TL.

4.5 Uber Example FPGA Design

4.5.1 Design Description

The Uber example FPGA design demonstrates functionality available on the Virtex 6 series of ADM-XRC boards. The design includes the following functional areas:

- Clock Generation (blk_clocks)

- Internal clock generation
- Internal reset generation
- External clock buffering (non-MGT sourced) and extraction (MGT sourced)

- MPTL Interface (mptl_if_target_wrap)

- OCP Direct Slave Channel (blk_direct_slave)

- Direct Slave address space splitter (adb3_ocp_reg_split)
- Simple test using host-accessible registers (blk_ds_simple_test)
- Clock frequency measurement using host-accessible registers (blk_ds_clk_read)
- XRM/PN4/PN6 GPIO test using host-accessible registers (blk_ds_io_test)
- Interrupt test using host-accessible registers (blk_ds_int_test)
- General purpose host-acessible registers including date and time stamps (blk_ds_info)
- Interface to BRAM in OCP DMA Block (adb3_ocp_cross_clk_dom)

- OCP DMA Channels (blk_dma)

- OCP DMA channel multiplex (adb3_ocp_mux)
- Interface to Block RAM (adb3_ocp_simple_bus_if)

- ChipScope Connection (optional)(blk_chipscope)

- ChipScope connection to OCP channels

A hierarchical diagram of the top level of the Uber example design is shown in **Figure 6, "Uber Design Top Level Hierarchy"**. A diagram of the package dependencies in then Uber example design is shown in **Figure 7, "Uber Design Package Dependencies"**

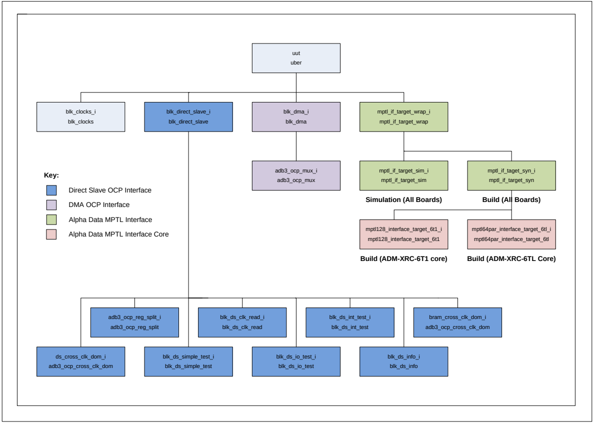


Figure 6: Uber Design Top Level Hierarchy

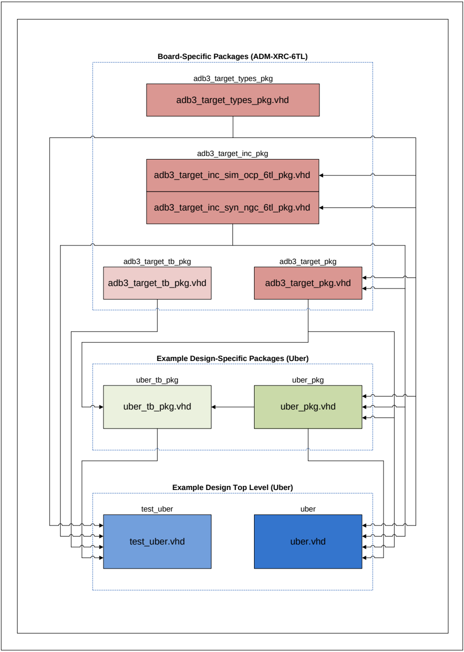

Board-Specific Packages (ADM-XRC-6TL)
adb3_target_types_pkg
adb3_target_types_pkg.vhd
adb3_target_inc_pkg
adb3_target_inc_sim_ocp_6tl_pkg.vhd
adb3_target_inc_syn_ngc_6tl_pkg.vhd
adb3_target_tb_pkg
adb3_target_tb_pkg.vhd
adb3_target_pkg
adb3_target_pkg.vhd
Example Design-Specific Packages (Uber)
uber_tb_pkg
uber_tb_pkg.vhd
uber_pkg
uber_pkg.vhd
Example Design Top Level (Uber)
test_uber
test_uber.vhd
uber
uber.vhd

Figure 7: Uber Design Package Dependencies

The Uber example FPGA design top level **uber.vhd** is located in **%ADMXRC3_SDK%\hdl\vhdl\examples\uber\common**. It consists of the following blocks:

- Clock generation block (blk_clocks)
- MPTL interface block (mptl_if_target_wrap)
- OCP Direct Slave interface block (blk_direct_slave)
- OCP DMA interface block (blk_dma)
- ChipScope connection block (optional)(blk_chipscope)

A top level block diagram of the Uber example design is shown in **Figure 8, "Uber Design Testbench And Top Level Block Diagram"**.

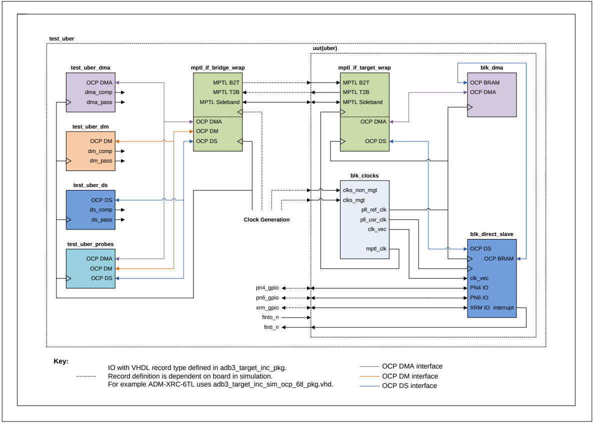


Figure 8: Uber Design Testbench And Top Level Block Diagram

4.5.1.1 Clock Generation Block

The clock and reset generation block is located in **%ADMXRC3_SDK%\fpga\examples\uber\common** in the file **blk_clocks.vhd**. It includes the following functional areas:

- Internal clock generation (MMCM)
- Internal reset generation
- External clock buffering (non-MGT sourced)
- External clock extraction (MGT sourced)
- MPTL interface clock generation

4.5.1.1.1 Internal Clock Generation (MMCM)

Two user clocks are generated from the MMCM: pll_ref_clk; and pll_usr_clk. These are used to drive the high speed (OCP DMA) and low speed (OCP Direct Slave) areas of the design. Refer to **Figure 9, "Uber Design Internal Clock Generation (MMCM)"**.

4.5.1.1.2 Internal reset generation

Two user resets are generated: pll_ref_rst; amd pll_usr_rst. These are used to drive the high speed (OCP DMA) and low speed (OCP Direct Slave) areas of the design. Refer to **Figure 9, "Uber Design Internal Clock Generation (MMCM)"**.

The resets are generated from their respective clocks: pll_ref_clk; and pll_usr_clk using the ADCOMMON library component **rst_sync**. Refer to Section 5 for a functional description.

4.5.1.1.3 External Clock Buffering (Non-MGT Sourced)

Non-MGT clock inputs are buffered. Clock support is dependent on board selected. Refer to **Figure 10, "Uber Design Clock Buffering/Extraction"**.

4.5.1.1.4 External Clock Extraction (MGT Sourced)

MGT clock inputs are converted from double-ended to single-ended and then buffered. The buffered clocks are connected to the clk_vec signal. The connection order is defined in the uber_pkg.vhd file. Clock support is dependent on board selected. Refer to **Figure 10, "Uber Design Clock Buffering/Extraction"**.

4.5.1.1.5 MPTL interface clock generation

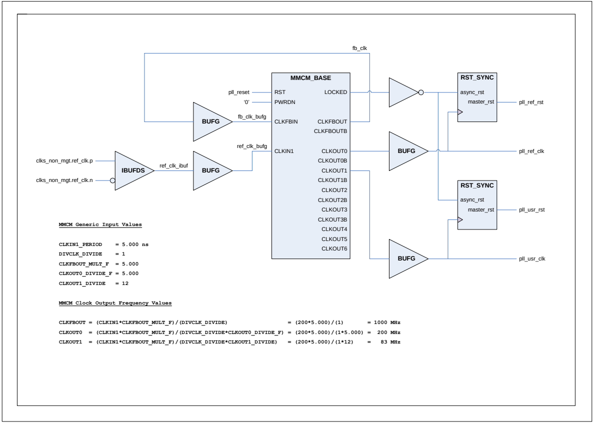
Refer to **Figure 10, "Uber Design Clock Buffering/Extraction"**.

The MPTL interface block requires an mptl_clk clock input. This is generated from an FPGA MGT clock input. The **mptl_clk** signal may be single or double ended depending on the board in use. Its type **mptl_clk_t**is defined in the board specific package **adb3_target_inc_pkg** which is located in the board directory in **%ADMXRC3_SDK%\fpga\common\adb3_target**.

During simulation, the **mptl_clk_t** record needs to contain both single and double ended clock record elements. Only the record elements appropriate to the board being simulated are driven.

During synthesis, the **mptl_clk_t** record need only contain the clock record elements specific to the board being built.

Two functions: **sgl_to_mptl_clk_t** and **dbl_to_mptl_clk_t** are provided in the board specific package **adb3_target_inc_pkg** to convert from the MGT clock input type to the mptl_clk_t type.



```
MMCM Generic Input Values

CLKIN1_PERIOD    = 5.000 ns
DIVCLK_DIVIDE    = 1
CLKFBOUT_MULT_F  = 5.000
CLKOUT0_DIVIDE_F = 5.000
CLKOUT1_DIVIDE   = 12

MMCM Clock Output Frequency Values

CLKFBOUT = (CLKIN1*CLKFBOUT_MULT_F)/(DIVCLK_DIVIDE)                  = (200*5.000)/(1)       = 1000 MHz
CLKOUT0  = (CLKIN1*CLKFBOUT_MULT_F)/(DIVCLK_DIVIDE*CLKOUT0_DIVIDE_F) = (200*5.000)/(1*5.000) =  200 MHz
CLKOUT1  = (CLKIN1*CLKFBOUT_MULT_F)/(DIVCLK_DIVIDE*CLKOUT1_DIVIDE)   = (200*5.000)/(1*12)    =   83 MHz
```

Figure 9: Uber Design Internal Clock Generation (MMCM)

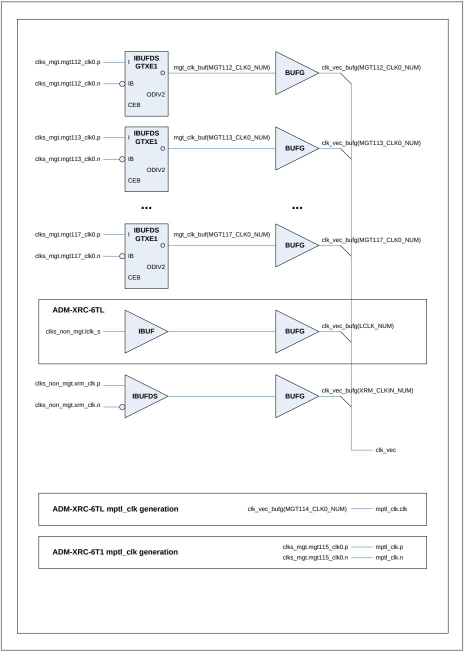


Figure 10: Uber Design Clock Buffering/Extraction

4.5.1.2 MPTL Interface Block

This block is implemented using the MPTL library component **mptl_if_target_wrap**. Refer to Section 5 for a functional description.

The Uber design output signal **mptl_target_configured_n** indicates that the FPGA is ready to communicate with the bridge via the MPTL interface. This output should be generated by combining the **mptl_if_target_wrap** output **mptl_target_configured_n** with any other FPGA signals indicating readiness. In the case of the Uber design, this output is ORed with each of the MMCM clock domain reset signals (active high). This ensures that communication is not initiated until the MMCM is locked and the resets are inactive.

4.5.1.3 OCP Direct Slave Interface Block

The OCP Direct Slave interface block is located in **%ADMXRC3_SDK%\fpga\examples\uber\common** in the file **blk_direct_slave.vhd**. It includes the following blocks:

- OCP address space splitter block (adb3_ocp_reg_split)
- Simple test block (blk_ds_simple_test)
- Clock read block (blk_ds_clk_read)
- GPIO test block (blk_ds_io_test)
- Interrupt test block (blk_ds_int_test)
- Info block (blk_ds_info)
- BRAM interface block (adb3_ocp_cross_clk_dom)

A block diagram of the OCP direct slave interface block is shown in **Figure 11, "Uber Direct Slave Block Diagram"**.

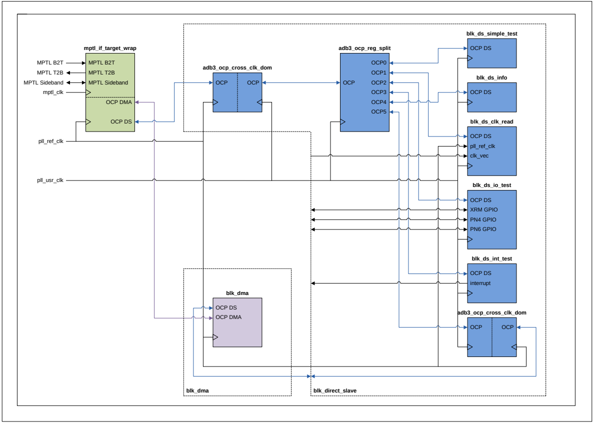


Figure 11: Uber Direct Slave Block Diagram

4.5.1.3.1 OCP Address Space Splitter Block

This block is implemented using the ADB3 OCP library component **adb3_ocp_reg_split**. Refer to Section 5 for a functional description.It splits the upstream OCP interface into multiple downstream OCP interfaces. The split is controlled by an address space range table which is defined using the **ADDR_RANGE_TABLE** constant in package **uber_pkg**.

The Uber example design OCP direct slave address space is split as follows:

Block	Type	Addr Range	Data Width
Simple	Registers	0x00000000-0x0000003F	32-bit
Clock Read	Registers	0x00000040-0x0000007F	32-bit
Interrupt	Registers	0x000000C0-0x000000FF	32-bit
Info	Registers	0x00000140-0x0000017F	32-bit
IO	Registers	0x00000200-0x0000027F	32-bit
BRAM Interface	BRAM	0x00080000-0x000FFFFF	128-bit

Table 4: Uber design Direct Slave Address Map

Note: Read transactions to undefined areas of the address space will return data containing 0x"DEADC0DE".

4.5.1.3.2 Simple Test Block

4.5.1.3.2.1 Description

The simple test block is located in **%ADMXRC3_SDK%\fpga\examples\uber\common** in the file **blk_ds_simple_test.vhd**. It consists of an OCP to parallel interface block, and a register section. The split OCP Direct Slave channel connects to the OCP to parallel interface block. This block is implemented using the ADB3 OCP library component **adb3_ocp_simple_bus_if**. Refer to Section 5 for a functional description.

The simple test block performs a nibble reverse function using its register interface.

4.5.1.3.2.2 Register Interface

The OCP to parallel interface block connects to the Simple registers. Write accesses are controlled by the write enable bus and/or the write signal. Read accesses are controlled by the read signal. The address bus is used to select the register to be accessed and data is transferred on the data busses.

The Simple test block implements registers in the Direct Slave OCP address space as follows:

Name	Type	Address
DATA	RW	0x00000000

Table 5: Uber Design Simple Test Block Address Map

Bits	Mnemonic	Type	Function
31:0	DATA	RW	Indicates the nibble-reversed version of the last data written.

Table 6: Uber Design Simple Test Block DATA Register

4.5.1.3.3 Clock Read Block

4.5.1.3.3.1 Description

The Clock Read Block is located in **%ADMXRC3_SDK%\fpga\examples\uber\common** in the file **blk_ds_clk_read.vhd**. It consists of clock frequency measurement blocks, an OCP to parallel interface block, and a register section. The clock frequency measurement blocks are implemented using the ADCOMMON library component **clock_speed**. Refer to Section 5 for a functional description. The split OCP Direct Slave channel connects to the OCP to parallel interface block. This block is implemented using the ADB3 OCP library component **adb3_ocp_simple_bus_if**. Refer to Section 5 for a functional description.

The clock read block allows the frequencies of all FPGA clocks to be read using its register interface.

4.5.1.3.3.2 Register Interface

The OCP to parallel interface block connects to the Clock Read registers. Write accesses are controlled by the write enable bus and/or the write signal. Read accesses are controlled by the read signal. The address bus is used to select the register to be accessed and data is transferred on the data busses.

The Clock Read block implements registers in the Direct Slave OCP address space as follows:

Name	Type	Address
SEL	RW	0x00000040
CTRL/STAT	RW	0x00000044
FREQ	RO	0x00000048

Table 7: Uber Design Clock Read Block Address Map

Bits	Mnemonic	Type	Function
31:5			Unused
4:0	SEL_CLK	RW	Controls selection of the FPGA clock data to be accessed using the STAT and FREQ registers. Selection is as follows: 00000 => pll_usr_clk (Internal user clock derived from ref_clk) 00001 => pll_ref_clk (Internal 200 MHz reference clock) 01100 => lclk (External) 01101 => xrm_clkin (External MGT clock) 10010 => mgt112_clk0 (External MGT clock) 10100 => mgt113_clk0 (External MGT clock) 10101 => mgt113_clk1 (External MGT clock) 10110 => mgt114_clk0 (External MGT clock) 11000 => mgt115_clk0 (External MGT clock) 11010 => mgt116_clk0 (External MGT clock) 11100 => mgt117_clk0 (External MGT clock)

Table 8: Uber Design Clock Read Block SEL Register

Bits	Mnemonic	Type	Function
31	CLR_UPDATE	RW	Write: controls frequency measurement updated flags: 1 = Clear all measurement updated flags. 0 = No action. Read: indicates selected frequency measurement update status: 1 = Measurement updated 0 = Measurement not updated.
30	CLK_VALID	RO	Indicates selected board clock valid status: 1 = Clock valid on this board. 0 = Clock not valid on this board.
29	CLK_RUNNING	RO	Indicates selected clock running status: 1 = Clock running 0 = Clock not running.
28:0			Unused

Table 9: Uber Design Clock Read Block CTRL/STAT Register

Bits	Mnemonic	Type	Function
31:0	FREQ	RO	Indicates selected clock frequency measurement in Hz.

Table 10: Uber Design Clock Read Block FREQ Register

4.5.1.3.4 GPIO Test Block

4.5.1.3.4.1 Description

The GPIO test block is located in **%ADMXRC3_SDK%\fpga\examples\uber\common** in the file **blk_ds_io_test.vhd**. It consists of an OCP to parallel interface block, and a register section. The split OCP Direct Slave channel connects to the OCP to parallel interface block. This block is implemented using the ADB3 OCP library component **adb3_ocp_simple_bus_if**. Refer to Section 5 for a functional description.

The GPIO test block controls the XRM, PN4, and PN6 GPIO bi-directional interfaces. Each bit of each interface may be controlled individually using its register interface.

4.5.1.3.4.2 Register Interface

The OCP to parallel interface block connects to the GPIO test block registers. Write accesses are controlled by the write enable bus and/or the write signal. Read accesses are controlled by the read signal. The address bus is used to select the register to be accessed and data is transferred on the data busses.

The GPIO test block implements registers in the Direct Slave OCP address space as follows:

Name	Type	Address
XRM_GPIO_DA_TRI	RW	0x00000200
XRM_GPIO_DA_DATA	RW	0x00000204
XRM_GPIO_DB_TRI	RW	0x00000208
XRM_GPIO_DB_DATA	RW	0x0000020C
XRM_GPIO_DC_TRI	RW	0x00000210
XRM_GPIO_DC_DATA	RW	0x00000214
XRM_GPIO_DD_TRI	RW	0x00000218
XRM_GPIO_DD_DATA	RW	0x0000021C
XRM_GPIO_CS_TRI	RW	0x00000220
XRM_GPIO_CS_DATA	RW	0x00000224

Table 11: Uber Design GPIO Test Block Address Map (continued on next page)

Name	Type	Address
PN4_GPIO_P_TRI	RW	0x00000228
PN4_GPIO_P_DATA	RW	0x0000022C
PN4_GPIO_N_TRI	RW	0x00000230
PN4_GPIO_N_DATA	RW	0x00000234
PN6_GPIO_MS_TRI	RW	0x00000238
PN6_GPIO_MS_DATA	RW	0x0000023C
PN6_GPIO_LS_TRI	RW	0x00000240
PN6_GPIO_LS_DATA	RW	0x00000244

Table 11: Uber Design GPIO Test Block Address Map

Bits	Mnemonic	Type	Function
31:0	DA_TRI	RW	Write: controls tristate enables of da_p(15:0), da_n(15:0) IO ports. Read: indicates tristate enables of da_p(15:0), da_n(15:0) IO ports.

Table 12: Uber Design GPIO Test Block XRM_GPIO_DA_TRI Register

Bits	Mnemonic	Type	Function
31:0	DA_DATA	RW	Write: controls data written to da_p(15:0), da_n(15:0) IO ports. Read: indicates data read from da_p(15:0), da_n(15:0) IO ports.

Table 13: Uber Design GPIO Test Block XRM_GPIO_DA_DATA Register

Bits	Mnemonic	Type	Function
31:0	DB_TRI	RW	Write: controls tristate enables of db_p(15:0), db_n(15:0) IO ports. Read: indicates tristate enables of db_p(15:0), db_n(15:0) IO ports.

Table 14: Uber Design GPIO Test Block XRM_GPIO_DB_TRI Register

Bits	Mnemonic	Type	Function
31:0	DB_DATA	RW	Write: controls data written to db_p(15:0), db_n(15:0) IO ports. Read: indicates data read from db_p(15:0), db_n(15:0) IO ports.

Table 15: Uber Design GPIO Test Block XRM_GPIO_DB_DATA Register

Bits	Mnemonic	Type	Function
31:0	DC_TRI	RW	Write: controls tristate enables of dc_p(15:0), dc_n(15:0) IO ports. Read: indicates tristate enables of dc_p(15:0), dc_n(15:0) IO ports.

Table 16: Uber Design GPIO Test Block XRM_GPIO_DC_TRI Register

Bits	Mnemonic	Type	Function
31:0	DC_DATA	RW	Write: controls data written to dc_p(15:0), dc_n(15:0) IO ports. Read: indicates data read from dc_p(15:0), dc_n(15:0) IO ports.

Table 17: Uber Design GPIO Test Block XRM_GPIO_DC_DATA Register

Bits	Mnemonic	Type	Function
31:0	DD_TRI	RW	Write: controls tristate enables of dd_p(15:0), dd_n(15:0) IO ports. Read: indicates tristate enables of dd_p(15:0), dd_n(15:0) IO ports.

Table 18: Uber Design GPIO Test Block XRM_GPIO_DD_TRI Register

Bits	Mnemonic	Type	Function
31:0	DD_DATA	RW	Write: controls data written to dd_p(15:0), dd_n(15:0) IO ports. Read: indicates data read from dd_p(15:0), dd_n(15:0) IO ports.

Table 19: Uber Design GPIO Test Block XRM_GPIO_DD_DATA Register

Bits	Mnemonic	Type	Function
17:0	CS_TRI	RW	Write: controls tristate enables of sa, sb, sc, sd, dd_cc_p, dd_cc_n, dc_cc_p, dc_cc_n, db_cc_p, db_cc_n, da_cc_p, da_cc_n IO ports. Read: indicates tristate enables of sa, sb, sc, sd, dd_cc_p, dd_cc_n, dc_cc_p, dc_cc_n, db_cc_p, db_cc_n, da_cc_p, da_cc_n IO ports.

Table 20: Uber Design GPIO Test Block XRM_GPIO_CS_TRI Register

Bits	Mnemonic	Type	Function
17:0	CS_DATA	RW	Write: controls data written to sa, sb, sc, sd, dd_cc_p, dd_cc_n, dc_cc_p, dc_cc_n, db_cc_p, db_cc_n, da_cc_p, da_cc_n IO ports. Read: indicates data read from sa, sb, sc, sd, dd_cc_p, dd_cc_n, dc_cc_p, dc_cc_n, db_cc_p, db_cc_n, da_cc_p, da_cc_n IO ports.

Table 21: Uber Design GPIO Test Block XRM_GPIO_CS_DATA Register

Bits	Mnemonic	Type	Function
PN4_GPIO_WIDTH-1:0	P_TRI	RW	Write: controls tristate enables of gpio_p(PN4_GPIO_WIDTH-1:0) IO ports.** Read: indicates tristate enables of gpio_p(PN4_GPIO_WIDTH-1:0) IO ports.**

Table 22: Uber Design GPIO Test Block PN4_GPIO_P_TRI Register

Bits	Mnemonic	Type	Function
PN4_GPIO_WIDTH-1:0	P_DATA	RW	Write: controls data written to gpio_p(PN4_GPIO_WIDTH-1:0) IO ports.** Read: indicates data read from gpio_p(PN4_GPIO_WIDTH-1:0) IO ports.**

Table 23: Uber Design GPIO Test Block PN4_GPIO_P_DATA Register

Bits	Mnemonic	Type	Function
PN4_GPIO_WIDTH-1:0	N_TRI	RW	Write: controls tristate enables of gpio_n(PN4_GPIO_WIDTH-1:0) IO ports.** Read: indicates tristate enables of gpio_n(PN4_GPIO_WIDTH-1:0) IO ports.**

Table 24: Uber Design GPIO Test Block PN4_GPIO_N_TRI Register

Bits	Mnemonic	Type	Function
PN4_GPIO_WIDTH-1:0	N_DATA	RW	Write: controls data written to gpio_n(PN4_GPIO_WIDTH-1:0) IO ports.** Read: indicates data read from gpio_n(PN4_GPIO_WIDTH-1:0) IO ports.**

Table 25: Uber Design GPIO Test Block PN4_GPIO_N_DATA Register

** The **PN4_GPIO_WIDTH** constant is defined in the **adb3_target_inc_pkg** package.

Bits	Mnemonic	Type	Function
PN6_GPIO_WIDTH-33:0	MS_TRI	RW	Write: controls tristate enables of gpio(PN6_GPIO_WIDTH-33:32) IO ports.++ Read: indicates tristate enables of gpio(PN6_GPIO_WIDTH-33:32) IO ports.++ Note: this register is applicable only on boards where PN6_GPIO_WIDTH > 32.

Table 26: Uber Design GPIO Test Block PN6_GPIO_MS_TRI Register

Bits	Mnemonic	Type	Function
PN6_GPIO_WIDTH-33:0	MS_DATA	RW	Write: controls data written to gpio(PN6_GPIO_WIDTH-33:32) IO ports.++ Read: indicates data read from gpio(PN6_GPIO_WIDTH-33:32) IO ports.++ Note: this register is applicable only on boards where PN6_GPIO_WIDTH > 32.

Table 27: Uber Design GPIO Test Block PN6_GPIO_MS_DATA Register

Bits	Mnemonic	Type	Function
PN6_GPIO_WIDTH-1:0	LS_TRI	RW	Write: controls tristate enables of gpio(PN6_GPIO_WIDTH-1:0) IO ports.++ Read: indicates tristate enables of gpio(PN6_GPIO_WIDTH-1:0) IO ports.++ Note: on boards where PN6_GPIO_WIDTH > 32 the register width is (31:0).

Table 28: Uber Design GPIO Test Block PN6_GPIO_LS_TRI Register

Bits	Mnemonic	Type	Function
PN6_GPIO_WIDTH-1:0	LS_DATA	RW	Write: controls data written to gpio(PN6_GPIO_WIDTH-1:0) IO ports.++ Read: indicates data read from gpio(PN6_GPIO_WIDTH-1:0) IO ports.++ Note: on boards where PN6_GPIO_WIDTH > 32 the register width is (31:0).

Table 29: Uber Design GPIO Test Block PN6_GPIO_LS_DATA Register

++ The **PN6_GPIO_WIDTH** constant is defined in the **adb3_target_inc_pkg** package.

4.5.1.3.5 Interrupt Test Block

4.5.1.3.5.1 Description

The Interrupt test block is located in **%ADMXRC3_SDK%\fpga\examples\uber\common** in the file **blk_ds_int_test.vhd**. It consists of an OCP to parallel interface block, a register section, and interrupt generation. The split OCP Direct Slave channel connects to the OCP to parallel interface block. This block is implemented using the ADB3 OCP library component **adb3_ocp_simple_bus_if**. Refer to Section 5 for a functional description.

The interrupt test block controls the generation of the **interrupt** output using its register interface.

4.5.1.3.5.2 Register Interface

The OCP to parallel interface block connects to the Interrupt registers. Write accesses are controlled by the write enable bus and/or the write signal. Read accesses are controlled by the read signal. The address bus is used to select the register to be accessed and data is transferred on the data busses.

The Interrupt test block implements registers in the Direct Slave OCP address space as follows:

Name	Type	Address
SET	WO	0x000000C0
CLEAR/STAT	RW	0x000000C4
MASK	RW	0x000000C8
ARM	RW	0x000000CC
COUNT	RW	0x000000D0

Table 30: Uber Design Interrupt Test Block Address Map

Bits	Mnemonic	Type	Function
31:0	SET	W1S	Controls the setting of individual bits in the STAT register. This will activate the interrupt output if these bits are not masked by the MASK register.

Table 31: Uber Design Interrupt Test Block SET Register

Bits	Mnemonic	Type	Function
31:0	CLEAR/STAT	RW1C	Write: controls the clearing of individual bits in the STAT register. Read: indicates the contents of the STAT register.

Table 32: Uber Design Interrupt Test Block CLEAR/STAT Register

Bits	Mnemonic	Type	Function
31:0	MASK	RW	Controls/indicates the masking (1) or enabling (0) of individual bits in the STAT register.

Table 33: Uber Design Interrupt Test Block MASK Register

Bits	Mnemonic	Type	Function
31:0	ARM	RW	A write to this register will force the FPGA interrupt output to its inactive state.

Table 34: Uber Design Interrupt Test Block ARM Register

Bits	Mnemonic	Type	Function
31:0	COUNT	RW	Write: if STAT register is zero, then write to the COUNT register to initialise. Read: indicates elapsed cycle count from STAT register becoming non-zero.

Table 35: Uber Design Interrupt Test Block COUNT Register

4.5.1.3.6 Info Block

4.5.1.3.6.1 Description

The Info block is located in **%ADMXRC3_SDK%\fpga\examples\uber\common** in the file **blk_ds_info.vhd**. It consists of an OCP to parallel interface block, and a register section. The split OCP Direct Slave channel connects to the OCP to parallel interface block. This block is implemented using the ADB3 OCP library component **adb3_ocp_simple_bus_if**. Refer to Section 5 for a functional description.

The Info block allows read access to its register interface.

4.5.1.3.6.2 Register Interface

The OCP to parallel interface block connects to the Info registers. Write accesses are controlled by the write enable bus and/or the write signal. Read accesses are controlled by the read signal. The address bus is used to select the register to be accessed and data is transferred on the data busses.

The Info block implements registers in the Direct Slave OCP address space as follows:

Name	Type	Address
DATE	RO	0x00000140
TIME	RO	0x00000144
SPLIT	RO	0x00000148
BASE	RO	0x0000014C
MASK	RO	0x00000150

Table 36: Uber Design Info Block Address Map

Bits	Mnemonic	Type	Function

31:0	DATE	RO	Date of build (DD/MM/YYYY) in BCD format where:
			DD = Day of month
			MM = Month of year
			YYYY = Year.

Table 37: Uber Design Info Block DATE Register

Bits	Mnemonic	Type	Function

31:0	TIME	RO	Time of build (HH/MM/SS/LL) in BCD format where:
			HH = Hour of day
			MM = Minute of hour
			SS = Second of minute
			LL = Millisecond of second.

Table 38: Uber Design Info Block TIME Register

Bits	Mnemonic	Type	Function
31:16	SPLIT	RO	Split block out of range address OCP write command count.
15:0	SPLIT	RO	Split block out of range address OCP read command count.

Table 39: Uber Design Info Block SPLIT Register

Bits	Mnemonic	Type	Function
31:0	BASE	RO	DMA BRAM base address in DMA OCP address space.

Table 40: Uber Design Info Block BASE Register

Bits	Mnemonic	Type	Function
31:0	MASK	RO	DMA BRAM mask address in DMA OCP address space.

Table 41: Uber Design Info Block MASK Register

4.5.1.3.7 BRAM Interface Block

The BRAM interface block connects the split Direct Slave OCP channel to the OCP DMA block BRAM. This block is implemented using the ADB3 OCP library component **adb3_ocp_cross_clk_dom**. Refer to Section 5 for a functional description.

4.5.1.4 OCP DMA Interface Block

The OCP DMA interface block is located in **%ADMXRC3_SDK%\fpga\examples\uber\common** in the file **blk_dma.vhd**. It includes the following blocks:

- OCP channel mux block (adb3_ocp_mux)
- OCP to parallel interface block (adb3_ocp_simple_bus_if)
- BRAM block (bram_single_wrap)

A block diagram of the OCP DMA interface block is shown in **Figure 12, "Uber DMA Block Diagram"**.

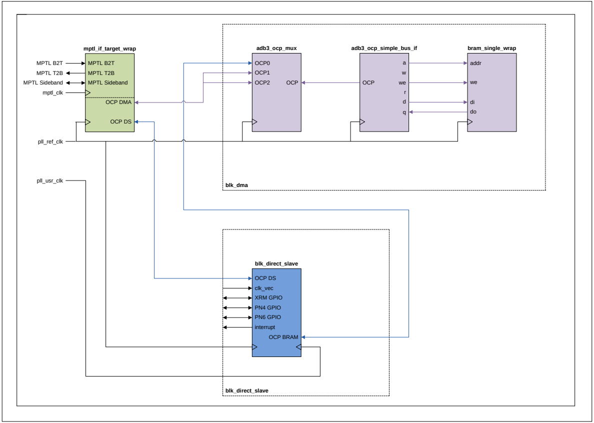


Figure 12: Uber DMA Block Diagram

4.5.1.4.1 OCP Channel Mux Block

The OCP channel mux block connects the DMA and Direct Slave OCP address spaces to the BRAM interface block. The mux block is implemented using the ADB3 OCP library component **adb3_ocp_mux**. Refer to Section 5 for a functional description.

4.5.1.4.2 OCP To Parallel Interface Block

The OCP to parallel interface block connects the OCP Channel Mux Block to the BRAM block. This block is implemented using the ADB3 OCP library component **adb3_ocp_simple_bus_if**. Refer to Section 5 for a functional description.

4.5.1.4.3 BRAM Block

The BRAM block instantiates 128 off the Xilinx™ BRAM_SINGLE_MACRO macro (36Kb x 1).

4.5.1.5 ChipScope Connection Block (optional)

The ChipScope connection block may be inserted in the Uber design if required. It is located in **%ADMXRC3_SDK%\fpga\common\chipscope** in the file **blk_chipscope.vhd**. It consists of three instantiations of the Xilinx™ ChipScope ILA block and a single instantiation of the Xilinx™ ChipScope ICON block. Each of the ILA blocks is connected to a single ChipScope connection block OCP channel.

Prior to the initial bitstream build of a design using the ChipScope connection block, the ChipScope ILA core **chipscope_ila.ngc** and ChipScope ICON core **chipscope_icon.ngc** need to be generated. These cores are generated using a script file. Examples are as follows:

To generate cores for Virtex 6 6vlx240t devices using windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\common\chipscope
gen_chipscope.bat 6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/common/chipscope
./gen_chipscope.bash 6vlx240t
```

4.5.2 Board Support

The Uber FPGA design is compatible with all Virtex 6 based boards.

4.5.3 Source Location

The Uber FPGA design is located in **%ADMXRC3_SDK%\hdl\vhdl\examples\uber**. Source files common to all boards are located in the **\common** directory. These include the design and testbench top levels.

For a complete list of the source files used during simulation, refer to the appropriate Modelsim macro file located in the board design directory, for example: **\admxrc6tl\simple-admxrc6tl.do** for the ADM-XRC-6TL.

For a complete list of the source files used during synthesis, refer to the appropriate XST project file located in the board design directory, for example: **\admxrc6t1\simple-admxrc6t1.prj** for the ADM-XRC-6T1.

4.5.4 Testbench Description

The Uber example FPGA design testbench **test_uber.vhd** is located in **%ADMXRC3_SDK%\hdl\vhdl\examples\uber\common**. Refer to **Figure 8, "Uber Design Testbench And Top Level Block Diagram"**.

The design testbench consists of the following functional areas:

- Clock generation

- Test Direct Slave interface (test_uber_ds)
- Test DMA interface (test_uber_dma)
- Bridge MPTL interface (mptl_if_bridge_wrap)
- OCP test probes (test_uber_probes)

The hierarchical structure of the design testbench is shown in **Figure 13, "Uber Design Testbench Hierarchy"**

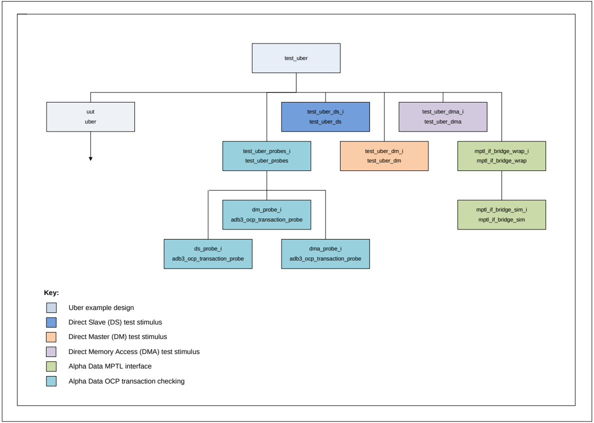


Figure 13: Uber Design Testbench Hierarchy

4.5.4.1 Clock Generation

4.5.4.1.1 Uber Example Design Clocks

The Uber example design requires clock inputs on the **clks_non_mgt** and **clks_mgt** signals.

- The **clks_non_mgt** input is a record defined in the adb3_target_inc_pkg package. Its definition is dependent on the board selected. It is connected to appropriate clocks generated by the testbench. Refer to the testbench file for connection information for each board.
- The **clks_mgt** input is a record defined in the adb3_target_inc_pkg package. Its definition is dependent on the board selected. It is connected to appropriate clocks generated by the testbench. Refer to the testbench file for connection information for each board.

4.5.4.1.2 Testbench Clocks

The Bridge MPTL Interface **mptl_if_bridge_wrap** input **ocp_clk** connection is dependent on the type of simulation selected.

- During OCP-OCP simulation, it must be driven by the same clock as the Simple example design MPTL Interface **mptl_if_target_wrap** input **ocp_clk**. This signal is transferred to the testbench using the **mptl_t2b.target_ocp_clk** record element.
- During OCP-MPTL-OCP simulation, it may be driven by an independent clock.

The Bridge MPTL Interface **mptl_if_bridge_wrap** input **mptl_clk** connection is dependent on the type of board selected. Refer to the testbench file for connection information for each board.

4.5.4.2 Test Direct Slave Interface

This function is implemented using the **test_uber_ds** entity. It connects to the **mptl_if_bridge_wrap** OCP direct slave interface and contains the following sections:

4.5.4.2.1 Simple Test

This section communicates with the Simple Test block registers as follows:

```
Write (32-bit), set DATA = 0x"cafeface"
Read  (32-bit), exp DATA = 0x"cafeface"
```

Section complete and pass/fail indications are returned using the **ds_comp.simple_complete** and **ds_pass.simple_passed** signals respectively.

4.5.4.2.2 Clock Read Test

This section communicates with the Clock Read block registers as follows:

```
Write (32-bit), set CLR_UPDATE = '1'

Write (32-bit), set SEL_CLK = PLL_USR_CLK_SEL
Read  (32-bit), exp CLR_UPDATE = '1'
Read  (32-bit), exp FREQ = 0x"00000053"

Write (32-bit), set SEL_CLK = PLL_REF_CLK_SEL
Read  (32-bit), exp CLR_UPDATE = '1'
Read  (32-bit), exp FREQ = 0x"000000C8"

Write (32-bit), set SEL_CLK = MGT113_CLK0_SEL
Read  (32-bit), exp CLR_UPDATE = '1'
Read  (32-bit), exp FREQ = Board Dependent
```

Section complete and pass/fail indications are returned using the **ds_comp.clock_complete** and **ds_pass.clock_passed** signals respectively.

4.5.4.2.3 Front IO (XRM GPIO) Test

This section communicates with the GPIO Test block registers as follows:

```
Write (32-bit), set XRM_GPIO_DD_DATA = 0x"76543210"
Write (32-bit), set XRM_GPIO_DD_TRI  = 0x"00000000"
Read  (32-bit), exp XRM_GPIO_DD_DATA = 0x"76543210"
```

Section complete and pass/fail indications are returned using the **ds_comp.frontio_complete** and **ds_pass.frontio_passed** signals respectively.

4.5.4.2.4 Rear IO (PN4/PN6 GPIO) Test

This section communicates with the GPIO Test block registers as follows:

```
Write (32-bit), set PN4_GPIO_P_DATA  = 0x"AABBCCDD"
Write (32-bit), set PN4_GPIO_N_DATA  = 0x"55443322"
Write (32-bit), set PN4_GPIO_P_TRI   = 0x"00000000"
Write (32-bit), set PN4_GPIO_N_TRI   = 0x"00000000"
Read  (32-bit), exp PN4_GPIO_P_DATA  = 0x"AABBCCDD"**
Read  (32-bit), exp PN4_GPIO_N_DATA  = 0x"55443322"**
Write (32-bit), set PN4_GPIO_P_TRI   = 0x"FFFFFFFF"
Write (32-bit), set PN4_GPIO_N_TRI   = 0x"FFFFFFFF"
Write (32-bit), set PN6_GPIO_MS_DATA = 0x"AAAABBBB"
Write (32-bit), set PN6_GPIO_LS_DATA = 0x"CCCCDDDD"
Write (32-bit), set PN6_GPIO_MS_TRI  = 0x"00000000"
Write (32-bit), set PN6_GPIO_LS_TRI  = 0x"00000000"
Read  (32-bit), exp PN6_GPIO_MS_DATA = 0x"AAAABBBB"++
Read  (32-bit), exp PN6_GPIO_LS_DATA = 0x"CCCCDDDD"++
Write (32-bit), set PN6_GPIO_MS_TRI  = 0x"FFFFFFFF"
Write (32-bit), set PN6_GPIO_LS_TRI  = 0x"FFFFFFFF"
```

** Data tested will be determined by the **PN4_GPIO_WIDTH** constant defined in the **adb3_target_inc_pkg** package.

++ Data tested will be determined by the **PN6_GPIO_WIDTH** constant defined in the **adb3_target_inc_pkg** package.

Section complete and pass/fail indications are returned using the **ds_comp.reario_complete** and **ds_pass.reario_passed** signals respectively.

4.5.4.2.5 Interrupt Test

This section communicates with the Interrupt Test block registers as follows:

```
Write (32-bit), set MASK  = 0x"00000000"
Read  (32-bit), exp MASK  = 0x"00000000"
Write (32-bit), set COUNT = 0x"FFFFFFFF"
Read  (32-bit), exp COUNT = 0x"FFFFFFFF"

for n in 0 to 31 loop
Write (32-bit), set SET = 2**n
wait for <b>linti_n</b> interrupt active
Read  (32-bit), exp CLEAR/STAT = 2**n
Write (32-bit), set CLEAR/STAT = 2**n
Write (32-bit), set ARM = Don't Care
end loop
Read  (32-bit), exp CLEAR/STAT = 0x"00000000"
```

Section complete and pass/fail indications are returned using the **ds_comp.int_complete** and **ds_pass.int_passed** signals respectively.

4.5.4.2.6 Info Test

This section communicates with the Info Test block registers as follows:

```
Read  (32-bit), exp DATE = 0x"13072010"
Read  (32-bit), exp TIME = 0x"18131232"
Read  (32-bit), exp BASE = 0x"00080000"
Read  (32-bit), exp MASK = 0x"0007FFFF"
```

Section complete and pass/fail indications are returned using the **ds_comp.info_complete** and **ds_pass.info_passed** signals respectively.

4.5.4.2.7 BRAM Test

This section communicates with the BRAM block in the OCP DMA Interface block as follows:

```
Write (32-bit), addr = 0x"00080000", data = 0x"2389EF45"
Read  (32-bit), addr = 0x"00080000",  exp = 0x"2389EF45"

Write (32-bit), addr = 0x"0007FFFC", data = 0x"369CF258"
Read  (32-bit), addr = 0x"0007FFFC",  exp = 0x"DEADCODE"

Write (32-bit), addr = 0x"00100000", data = 0x"258BE147"
Read  (32-bit), addr = 0x"00100000",  exp = 0x"DEADCODE"

Write (32-bit), addr = 0x"000F000C", data = 0x"147AD036"
Read  (32-bit), addr = 0x"000F000C",  exp = 0x"147AD036"
```

Section complete and pass/fail indications are returned using the **ds_comp.bram_complete** and **ds_pass.bram_passed** signals respectively.

4.5.4.3 Test DMA Interface

This function is implemented using the **test_uber_dma** entity. It connects to the **mptl_if_bridge_wrap** OCP DMA interface and contains the following sections:

4.5.4.3.1 DMA Write Channel Process

This process communicates with the BRAM block in the OCP DMA Interface block as follows:

```
for n in 0 to (DMA_SIZE/96)-1 loop
Write (96-byte), addr = DMA_ADDR_WR, data = 96 bytes incrementing patten
end loop
```

Section complete and pass/fail indications are returned using the **dma_comp.dma_write_complete** and **dma_pass.dma_write_passed** signals respectively.
The **DMA_SIZE** and **DMA_ADDR_WR** constants are defined in the **uber_tb_pkg** package.

4.5.4.3.2 DMA Read Channel Process

This process communicates with the BRAM block in the OCP DMA Interface block as follows:

```
for n in 0 to (DMA_SIZE/64)-1 loop
Read (64-byte), addr = DMA_ADDR_RD,  exp = 64 bytes incrementing patten
end loop
```

Section complete and pass/fail indications are returned using the **dma_comp.dma_read_complete** and **dma_pass.dma_read_passed** signals respectively.
The **DMA_SIZE** and **DMA_ADDR_RD** constants are defined in the **uber_tb_pkg** package.

4.5.4.4 Bridge MPTL interface

This function is implemented using the MPTL library component **mptl_if_bridge_wrap**. Refer to Section 5 for a functional description.

The **mptl_if_bridge_wrap** component output **bridge_gtp_online_n** is combined with the Uber example design output **target_gtp_online_n** to produce the **mptl_online_long** signal. This indicates that the MPTL interface is active and stable. This signal is monitored and will terminate the simulation if it goes inactive.

4.5.4.5 OCP test probes

This function is implemented using the **test_uber_probes** entity. It monitors each of the OCP interfaces for transaction errors using the ADB3 Probe library component **adb3_ocp_transaction_probe_sim**. Refer to Section 5 for a functional description.

4.5.5 Design Simulation

Modelsim macro files are located in **%ADMXRC3_SDK%\fpga\examples\uber** in each of the board design directories. For example **\admxrc6tl\uber-admxrc6tl.do** for the ADM-XRC-6TL.

Modelsim simulation is initiated using the **vsim** command with the appropriate macro file. For example, to perform a modelsim simulation using windows and the ADM-XRC-6TL, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
vsim -do "uber-admxrc6tl.do"
```

Expected simulation results are shown below.

4.5.5.1 Date/Time Package Generation Results

Before compiling HDL and initiating simulation, the macro file runs a script to generate the gen_today_pkg.vhd file. This file contains HDL constant definitions containing the date and time at which the script was run. The script executed will be **gen_today_pkg.bat** using windows, and **gen_today_pkg.bash** using Linux. Script output will be similar to the following example:

```
# Running gen_today_pkg.bat...
#
#
# Date Format: dd/MM/yyyy
#   Date Read: 20/07/2010
#  Date Stamp: 20072010
#
# Time Format: HH:mm:ss
#   Time Read: 15:57:14.49
#  Time Stamp: 15571449
#
# Output File: .\common\today_pkg.vhd
#
# Batch file complete
```

4.5.5.2 Initialisation Results

Modelsim output during initialisation of simulation will be similar to the following example:

```
# ** Note: Board Type : adm_xrc_6tl
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
# ** Note: Target Use : sim_ocp
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
# ** Note: Waiting for MPTL online....
#    Time: 0 ps  Iteration: 0  Instance: /test_simple
```

4.5.5.3 Test Direct Slave Block Test Results

4.5.5.3.1 Simple Test Results

Modelsim output during simulation will be similar to the following example:

```
# ** Note: Wrote simple WDATA 4 bytes 0xCAFEFACE with enable 0b1111 to byte address 0
#    Time: 1830 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read simple RDATA 4 bytes 0xECAFEFAC from byte address 0
#    Time: 2132500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Test Simple completed: PASSED.
```

4.5.5.3.2 Clock Read Test Results

Modelsim output during simulation will be similar to the following example:

```
# ** Note: Wrote Clear All CTRL 4 bytes 0x80000000 with enable 0b1111 to byte address 68
#    Time: 2305 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PLL_USR_CLK_SEL SEL 4 bytes 0x00000000 with enable 0b1111 to byte address 64
#    Time: 2315 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PLL_USR_CLK FREQ 4 bytes 0x00000053 from byte address 72
#    Time: 3872500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 83 or 84 MHz
#    Time: 3872500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual freq = 83 MHz
#    Time: 3872500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PLL_REF_CLK_SEL SEL 4 bytes 0x00000001 with enable 0b1111 to byte address 64
#    Time: 3880 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PLL_REF_CLK FREQ 4 bytes 0x000000C8 from byte address 72
#    Time: 4412500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 200 MHz
#    Time: 4412500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual freq = 200 MHz
#    Time: 4412500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote MGT113_CLK0_SEL SEL 4 bytes 0x00000014 with enable 0b1111 to byte address 64
#    Time: 4420 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read MGT113_CLK0 FREQ 4 bytes 0x000000FA from byte address 72
#    Time: 4952500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Expected freq = 250 MHz
#    Time: 4952500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Actual freq = 250 MHz
#    Time: 4952500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Test Clock Read completed: PASSED.
#    Time: 4952500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

4.5.5.3.3 Front IO (XRM GPIO) Test Results

Modelsim output during simulation will be similar to the following example:

```
# ** Note: Wrote XRM_GPIO_DA DATA 4 bytes 0x76543210 with enable 0b1111 to byte address 540
#    Time: 5125 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote XRM_GPIO_DA TRI 4 bytes 0x00000000 with enable 0b1111 to byte address 536
#    Time: 5135 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read XRM_GPIO_DA DATA 4 bytes 0x76543210 from byte address 540
#    Time: 5622500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote XRM_GPIO_DA TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte address 536
#    Time: 5630 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Test Front IO completed: PASSED.
#    Time: 5630 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

4.5.5.3.4 Rear IO (PN4/PN6 GPIO) Test Results

Modelsim output during simulation will be similar to the following example:

```
# ** Note: Wrote PN4_GPIO_P DATA 4 bytes 0xAABBCCDD with enable 0b1111 to byte address 556
#    Time: 5640 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN4_GPIO_N DATA 4 bytes 0x55443322 with enable 0b1111 to byte address 564
#    Time: 5650 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN4_GPIO_P TRI 4 bytes 0x00000000 with enable 0b1111 to byte address 552
#    Time: 5660 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN4_GPIO_N TRI 4 bytes 0x00000000 with enable 0b1111 to byte address 560
#    Time: 5670 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PN4_GPIO_P DATA 4 bytes 0xAABBCCDD from byte address 556
#    Time: 6212500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PN4_GPIO_N DATA 4 bytes 0x55443322 from byte address 564
#    Time: 6452500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN4_GPIO_P TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte address 552
#    Time: 6460 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN4_GPIO_N TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte address 560
#    Time: 6470 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_MS WDATA 4 bytes 0xAAAABBBB with enable 0b1111 to byte address 572
#    Time: 6480 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_LS WDATA 4 bytes 0xCCCCDDDD with enable 0b1111 to byte address 580
#    Time: 6490 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_MS TRI 4 bytes 0x00000000 with enable 0b1111 to byte address 568
#    Time: 6500 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_LS TRI 4 bytes 0x00000000 with enable 0b1111 to byte address 576
#    Time: 6510 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PN6_GPIO_MS RDATA 4 bytes 0xXXXXXXXB from byte address 572
#    Time: 7122500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read PN6_GPIO_LS RDATA 4 bytes 0xCCCCDDDD from byte address 580
#    Time: 7352500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_MS TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte address 568
#    Time: 7360 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote PN6_GPIO_LS TRI 4 bytes 0xFFFFFFFF with enable 0b1111 to byte address 576
#    Time: 7370 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Test Rear IO completed: PASSED.
#    Time: 7370 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

4.5.5.3.5 Interrupt Test Results

Modelsim output during simulation will be similar to the following example:

```
# ** Note: Wrote Interrupt MASK 4 bytes 0x00000000 with enable 0b1111 to byte address 200
#    Time: 7645 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Interrupt MASK 4 bytes 0x00000000 from byte address 200
#    Time: 7937500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote Interrupt COUNT 4 bytes 0xFFFFFFFF with enable 0b1111 to byte address 208
#    Time: 7945 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Interrupt COUNT 4 bytes 0xFFFFFFFF from byte address 208
#    Time: 8237500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Interrupt Monitor: Detected falling edge on linti_n
#    Time: 8361 ns  Iteration: 12  Instance: /test_uber
# ** Note: Interrupt Handler: Cleared interrupt(s), masked STAT = 0x00000001
#    Time: 8680 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Interrupt Monitor: Detected falling edge on linti_n
#    Time: 8949 ns  Iteration: 12  Instance: /test_uber
# ** Note: Interrupt Handler: Cleared interrupt(s), masked STAT = 0x00000002
#    Time: 9265 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i

...

# ** Note: Interrupt Monitor: Detected falling edge on linti_n
#    Time: 26349 ns  Iteration: 12  Instance: /test_uber
# ** Note: Interrupt Handler: Cleared interrupt(s), masked STAT = 0x80000000
#    Time: 26665 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Interrupt STAT 4 bytes 0x00000000 from byte address 196
#    Time: 27032500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Test Interrupt completed: PASSED.
```

4.5.5.3.6 Info Test Results

Modelsim output during simulation will be similar to the following example:

```
# ** Note: Read Info DATE 4 bytes 0x13072010 from byte address 320
#    Time: 27427500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info TIME 4 bytes 0x18131232 from byte address 324
#    Time: 27657500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info BASE 4 bytes 0x00080000 from byte address 332
#    Time: 27897500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read Info MASK 4 bytes 0x0007FFFF from byte address 336
#    Time: 28137500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Test Info completed: PASSED.
```

4.5.5.3.7 BRAM Test Results

Modelsim output during simulation will be similar to the following example:

```
# ** Note: Wrote BRAM Addr base 4 bytes 0x2389EF45 with enable 0b1111 to byte address 524288
#    Time: 28145 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read BRAM Addr base 4 bytes 0x2389EF45 from byte address 524288
#    Time: 28422500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote OOR Addr base-4 4 bytes 0x369CF258 with enable 0b1111 to byte address 524284
#    Time: 29005 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read OOR Addr base-4 4 bytes 0xDEADC0DE from byte address 524284
#    Time: 29202500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote OOR Addr top+1 4 bytes 0x258BE147 with enable 0b1111 to byte address 1048576
#    Time: 29210 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read OOR Addr top+1 4 bytes 0xDEADC0DE from byte address 1048576
#    Time: 29407500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Wrote BRAM Addr top 4 bytes 0x147AD036 with enable 0b1111 to byte address 1048572
#    Time: 29640 ns  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Read BRAM Addr top 4 bytes 0x147AD036 from byte address 1048572
#    Time: 29912500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
# ** Note: Test BRAM completed: PASSED.
#    Time: 29912500 ps  Iteration: 13  Instance: /test_uber/test_uber_ds_i
```

4.5.5.4 Test DMA Block Test Results

Modelsim output during simulation will be similar to the following example:

```
# ** Note: DMA write process started
#    Time: 1820 ns  Iteration: 14  Instance: /test_uber/test_uber_dma_i
# ** Note: DMA read process started
#    Time: 1895 ns  Iteration: 13  Instance: /test_uber/test_uber_dma_i
# ** Note: DMA write process completed
#    Time: 5530 ns  Iteration: 13  Instance: /test_uber/test_uber_dma_i
# ** Note: 4032 bytes transferred.
#    Time: 5530 ns  Iteration: 13  Instance: /test_uber/test_uber_dma_i
# ** Note: DMA read process completed
#    Time: 34225 ns  Iteration: 13  Instance: /test_uber/test_uber_dma_i
# ** Note: 4032 bytes transferred with 0 data error(s)
#    Time: 34225 ns  Iteration: 13  Instance: /test_uber/test_uber_dma_i
# ** Note: Test DMA completed: PASSED.
#    Time: 34225 ns  Iteration: 13  Instance: /test_uber/test_uber_dma_i
```

4.5.5.5 Completion Results

Modelsim output on successful completion of simulation will be similar to the following example:

```
# ** Failure: Test of design UBER completed: PASSED.
#    Time: 34230 ns  Iteration: 15  Process: /test_uber/test_results_p File: ../common/test_uber.vhd
# Break in Process test_results_p at ../common/test_uber.vhd line 328
# Simulation Breakpoint: Break in Process test_results_p at ../common/test_uber.vhd line 328
# MACRO ./uber-admxrc6tl.do PAUSED at line 74
```

4.5.6 Bitstream Build

A makefile is provided for building all bitstreams, or a specific board/device bitstream, for the Uber FPGA example design. It is located in the **%ADMXRC3_SDK\hdl\vhdl\examples\uber** directory. In order to use the re-built bitstream with the example applications, they must be copied to the **%ADMXRC3_SDK%\bit\uber** directory. This can be performed automatically using the **install** makefile option. A "clean up" of the files produced by the build process can be performed using the **clean** makefile option. Examples are as follows:

To perform a build of all Uber design bitstreams using windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake clean all
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make clean all
```

To perform a build and install the resulting bitstreams using windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake clean install
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make clean install
```

To perform a build for an ADM-XRC-6TL board fitted with an 6VLX240T device using windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake bit_admxrc6tl_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make bit_admxrc6tl_6vlx240t
```

To perform a build and install for an ADM-XRC-6TL board fitted with an 6VLX240T device using windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake inst_admxrc6tl_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make inst_admxrc6tl_6vlx240t
```

To perform a clean for an ADM-XRC-6TL board fitted with an 6VLX240T device using windows, start a shell and issue the following commands:

```
cd /d %ADMXRC3_SDK%\hdl\vhdl\examples\uber
nmake clean_admxrc6tl_6vlx240t
```

Similarly using Linux, start a shell and issue the following commands:

```
cd $ADMXRC3_SDK/hdl/vhdl/examples/uber
make clean_admxrc6tl_6vlx240t
```

The full path and filename of bitstreams built using Windows will be:

```
%ADMXRC3_SDK%\hdl\vhdl\examples\uber\output\<design>-<board>-<device>.bit
```

The full path and filename of bitstreams built using Linux will be:

```
$ADMXRC3_SDK/hdl/vhdl/examples/uber/output/<design>-<board>-<device>.bit
```

4.5.6.1 Date/Time Package Generation Results

If XST is required to be run during bitstream build, the makefile runs a script to generate the gen_today_pkg.vhd file. This file contains HDL constant definitions containing the date and time at which the script was run. The script executed will be **gen_today_pkg.bat** using windows, and **gen_today_pkg.bash** using Linux. Script output will be similar to the following example:

```
# Running gen_today_pkg.bat...
#
#
# Date Format: dd/MM/yyyy
#   Date Read: 20/07/2010
#  Date Stamp: 20072010
#
# Time Format: HH:mm:ss
#   Time Read: 15:57:14.49
#  Time Stamp: 15571449
#
# Output File: .\common\today_pkg.vhd
#
# Batch file complete
```

4.5.7 ISE Constraint Files

Constraint files for Uber design bitstream files using ISE are provided. These files are located in **%ADMXRC3_SDK%\hdl\vhdl\examples\uber** in each of the board design directories, for example **\admxrc6tl\uber-admxrc6tl.ucf** for the ADM-XRC-6TL.

5 Common HDL components

The ADM-XRC Gen 3 SDK provides a number of HDL components that are used in the example FPGA designs. These components may also be used in customer FPGA designs, and this section details their interfaces and usage.

5.1 ADB3 OCP Library

TBA

5.1.1 adb3_ocp_pkg Package

TBA

5.1.2 adb3_ocp_cross_clk_dom Component

TBA

5.1.3 adb3_ocp_mux Component

TBA

5.1.4 adb3_ocp_reg_split Component

TBA

5.1.5 adb3_ocp_simple_bus_if Component

TBA

5.2 MPTL Library

TBA

5.2.1 mptl_pkg Package

TBA

5.2.2 mptl_if_bridge_wrap Component

TBA

5.2.2.1 OCP-OCP Simulation

TBA

5.2.2.2 OCP-MPTL-OCP Simulation

TBA

5.2.2.3 Synthesis

TBA

5.2.3 mptl_if_target_wrap Component

5.2.3.1 OCP-OCP Simulation

TBA

5.2.3.2 OCP-MPTL-OCP Simulation

TBA

5.2.3.3 Synthesis

TBA
TBA

5.3 ADB3 Target Library

TBA

5.3.1 adb3_target_types_pkg Package

TBA

5.3.2 adb3_target_pkg Package

TBA

5.3.3 adb3_target_tb_pkg Package

TBA

5.4 ADB3 Probe Library

TBA

5.4.1 adb3_probe_pkg Package

TBA

5.4.2 adb3_ocp_transaction_probe_sim Component

TBA

5.5 ADCOMMON Library

TBA

5.5.1 cdc_pkg Package

TBA

5.5.2 clock_speed_pkg Package

TBA

5.5.2.1 clock_speed Component

TBA

5.5.3 rst_pkg Package

TBA

5.5.3.1 rst_sync Component

TBA

6 FPGA design guide

This section provides guidelines for FPGA designs targetting third generation Alpha Data hardware.

6.1 ADB3 OCP Protocol Reference

6.1.1 Introduction

This document describes the ADB3 OCP Protocol used by the MPTL Interface block in the target FPGA. The protocol is based on the OCP-IP standard which allows flexible connection between modules which share the OCP protocol. Differences in the implemented subset of OCP signals can be overcome by tieing the mismatching signals to default values. In general however most designs should simply stick to using the ADB3 OCP Protocol for module design, unless exisiting OCP-IP with a different protocol is included.

OCP-IP Protocols in general allow interfacing between 2 IP modules, with one module the master (in control of the transactions) and one module the slave. Each OCP-IP Protocol must have at least a command (Cmd) signal however the definition of other sideband signals is fairly flexible. The main groupings of signals used in the ADB3 OCP protocol are a Command Group, synchronous to the Cmd signal, and Data transfer groups both from Master to Slave (Write) and Slave to Master (Read Response). Each of these groupings is acknowledged independently allowing the flow to be controlled.

Signal	Group	Type	Description
Cmd	Command	OCP Cmd	Idle,Write or Read
Addr	Command	64 bit std_logic_vector	Address
BurstLength	Command	12 bit std_logic_vector	Length of transfer
Data	Data	128 bit std_logic_vector	Write Data to Slave
DataByteEn	Data	16 bit std_logic_vector	Byte Enables for Data
DataValid	Data	std_logic	Qualifier for Data
RespAccept	Response	std_logic	Flow Control for response
Tag	Command	8 bit std_logic_vector	Tag for Read response data

Table 42: ADB3 OCP Master Signals

Signal	Group	Type	Description
CmdAccept	Command	std_logic	Flow Control for commands
DataAccept	Data	std_logic	Flow Control for write Data
Data	Response	128 bit std_logic_vector	Response Data to Master
Resp	Response	OCP Resp	Qualifier for Response Data
Tag	Response	8 bit std_logic_vector	Tag for Read response data

Table 43: ADB3 OCP Slave Signals

In the VHDL code, the master and slave are grouped into record types to simplify the high level abstract system design, allowing the upper level modules to be specified in terms of module master to slave connections, rather than routing individual siganls

6.1.2 Timing Diagrams

This section contains timing diagrams for most common transactions and highlight the main operation of the protocol.

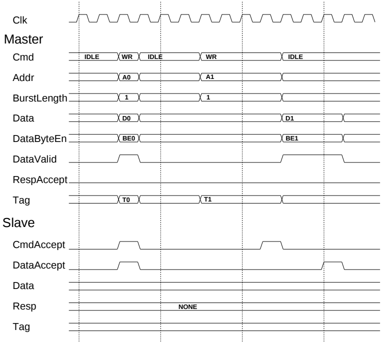


Figure 14: Single Beat Write

Figure 14, "Single Beat Write" shows 2 single beat write commands. The address, burst length and tag are all presented at the same time as the Cmd is set to Write. The Cmd is acknowledged within 1 clock cycle in the first case and so the Cmd is returned to Idle after a single clock cycle. In the first case, the Data and Byte Enables are asserted and accepted also in the same clock cycle. In the second case, the Write command is not accepted until the 4th cycle after it is asserted (possible due to teh Slave being busy). The master in this case also does not assert the Data Valid signal until after the Cmd. The data accept is also not accepted immediately and therefore the Data Valid must remain high until the data beat is accepted. All these cases constitute legal OCP transfers with the protocol.

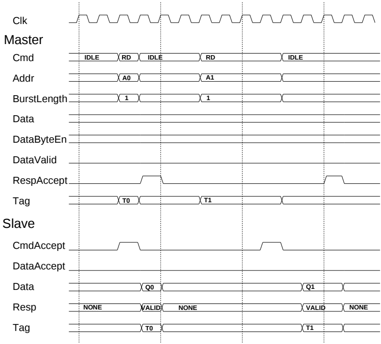


Figure 15: Single Beat Read

Figure 15, "Single Beat Read" shows 2 single beat read commands. in the first case the read request is immediately accepted. The slave responds with a response (Q0) on the following clock cycle. The Tag send with the read command is returned with the response. The second example shows a delayed command accept, a delayed response and a delayed response accept, all of which are legal with the protocol.

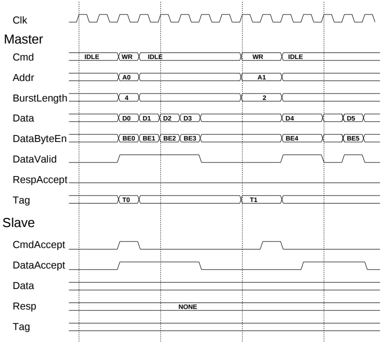


Figure 16: Burst Write

Figure 16, "Burst Write" shows 2 burst writes. A single command is issued for multiple data word transfers. The command protocol operates in exactly the same manner as for single beat transfers. Multiple data transfers occur for each command. Data transfers only occur when both DataValid and DataAccept are asserted. The master must wait on DataAccept being asserted before presenting the next data word. The slave must check that DataValid is asserted when receiving data. The slave may assert DataAccept even if DataValid is not asserted.

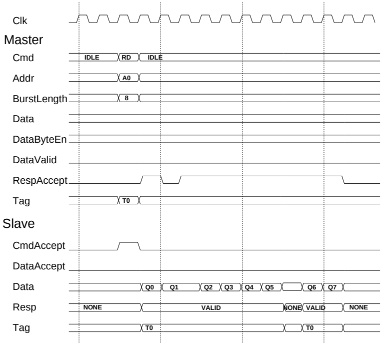


Figure 17: Burst Read

Figure 17, "Burst Read" shows a read burst. The response should be held valid and the read tag returned by the slave for all data transfers. Each data transfer required the Response to be Valid and RespAccect to be asserted.

7 The ADMXRC3 API

The ADMXRC3 API is the application programming interface that applications, including the ones in this SDK, use to communicate with third generation Alpha Data hardware. This API is documented in the **ADMXRC3 API Specification**.

Revision History:

Date	Revision	Nature of Change
20/05/2010	1.0	Initial version
26/07/2010	1.1	Updated for release 1.1.0 Added SDK structure diagram. Added information about example applications.