ADM-XRC-KU1 Interrupt Test FPGA Design

Release: 1.1.0

Document Revision: 1.1
5 Sep 2022

	Head Office	US Office
Address:	Suite L4A, 160 Dundee Street, Edinburgh, EH11 1DQ, UK	10822 West Toller Drive, Suite 250 Littleton, CO 80127
Telephone:	+44 131 558 2600	(303) 954 8768
Fax:	+44 131 558 2700	(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

List of Tables

List of Figures

1 Introduction

Supported Vivado versions

This version of the **ADM-XRC-KU1 Interrupt Test FPGA Design** can be built with Vivado 2018.3 or later.

As of writing, Vivado 2022.1 is the latest release and is recommended. Alpha Data cannot guarantee that this FPGA design will be fully compatible with future releases of Vivado.

This example FPGA design demonstrates how a program running on the host can consume notifications that the FPGA requires attention, which can be considered an abstraction of a PCI Express message-signaled interrupt, and how to implement a simple interrupt controller within the FPGA so that multiple sources of interrupt can be supported. There are several use cases for this capability, including:

- Notifying a host program that the FPGA has finished processing a block of data.
- Notifying a host program that the FPGA has finished receiving or transmitting a packet of data on a physical I/O interface.
- Notifying a host program that an exceptional (error) condition has arisen.

Although the FPGA has a single pin for generating an interrupt to the host, in any practical application, the host program must take some action after being notified that the FPGA requires attention. This means that the FPGA design must include a host interface so that the host program can read and write registers. In this FPGA design, the host interface is implemented using the Alpha Data **ADM-XRC-KU1-HSAXI** (Host Serial AXI4) IP.

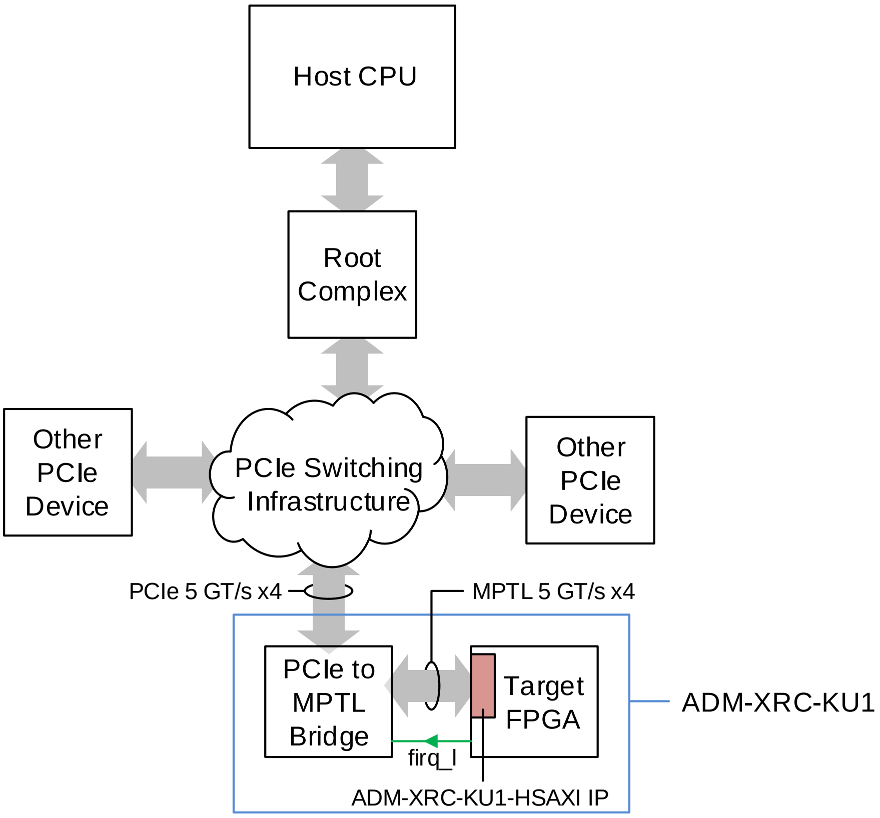


Figure 1 : The ADM-XRC-KU1 within a system

Within the ADM-XRC-KU1, the PCIe to MPTL Bridge performs a fixed function, namely to permit the target FPGA to be reconfigured without generating PCI Express errors that are fatal to the system; it is not user-programmable. The target FPGA, on the other hand, is user-programmable and may be reconfigured at will.

The **firq_l** signal (green) is the FPGA to Host interrupt signal. The Bridge FPGA converts a falling edge on this signal into a hardware interrupt (a PCI Express message-signaled interrupt), which is handled by Alpha Data's **ADB3 Driver**. The ADB3 Driver is a device driver for the ADM-XRC-KU1, provided by Alpha Data, with a user-mode application-programming interface (API) named the **ADMXRC3 API**. The ADB3 Driver converts hardware interrupts into "notifications" which can be consumed via the ADMXRC3 API.

The **ADM-XRC-KU1-HSAXI** (Host Serial AXI4) IP is instantiated in the target FPGA and has several interfaces:

- The MPTL (Multiplexed Packet Transport Link) interface; this is an off-chip interface to the Bridge FPGA, implemented using MGTs (Multi-Gigabit Transceivers).
- The Direct Slave interface; this is an AXI4 memory-mapped (AXI4-MM) master interface through which reads and writes originating on the host CPU can be performed. It is named "Direct Slave" because, from the point of view of the host CPU, the target FPGA is a slave.

  This interface is appropriate for random access, by the host CPU, to registers implemented in the target FPGA.
- Two DMA channels, which are AXI4 memory-mapped (AXI4-MM) slave interfaces through which reads and writes originating within the PCIe to MPTL Bridge are performed.

  These interfaces are appropriate for bulk data transfer between host memory and the target FPGA, but are not used in this example FPGA design.

This example FPGA design, then, demonstrates the following techniques:

- Instantiating ADM-XRC-KU1-HSAXI so that a host program can read and write registers within the FPGA.
- Creating a simple interrupt controller whose registers can be read and written by a host program.
- Handling multiple sources of interrupts that originate within the FPGA; in this example, there are two interrupt sources, consisting of (a) a periodic timer interrupt and (b) a TEST register which generates an interrupt when written to.

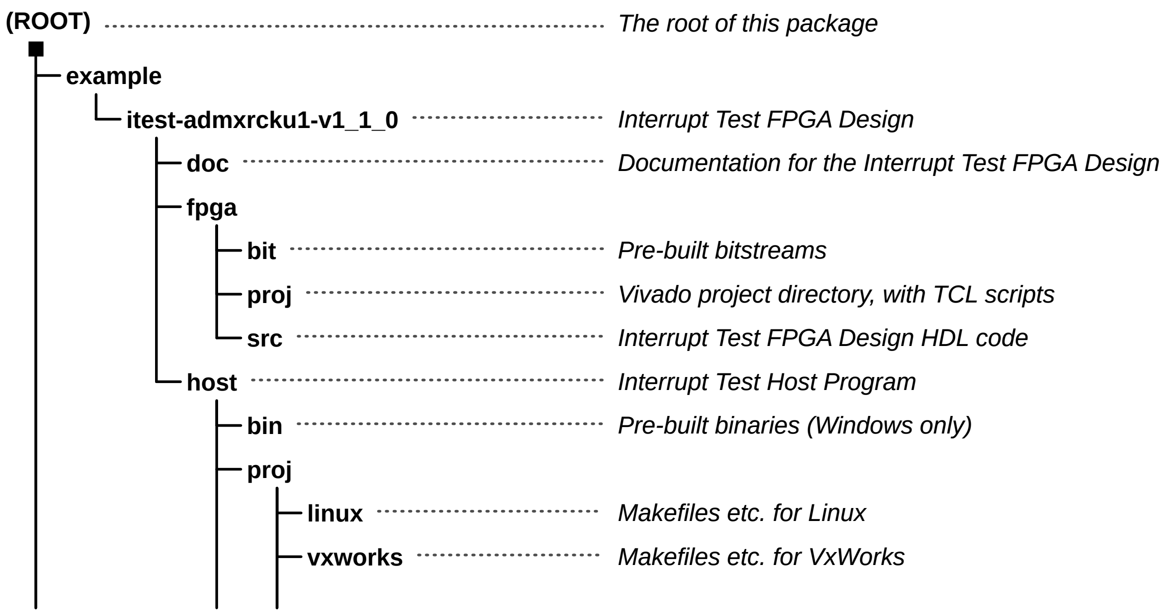
Tcl scripts are provided for generating Vivado projects for the FPGA design, and for building the FPGA design. Refer to Section 3 for details. Using the Simple FPGA design in hardware is described in Section 6.

To exercise the FPGA design, a demonstration program that runs on the host is provided, with source code. For details of building this program, refer to Section 5.

Using the FPGA design in hardware with the demonstration program is described in Section 6.

1.1 Structure of this package

The files and folders making up the Interrupt Test FPGA Design are organized as in Figure 2 below:


(ROOT) The root of this package
example
itest-admxrcku1-v1_1_0 Interrupt Test FPGA Design
doc Documentation for the Interrupt Test FPGA Design
fpga
bit Pre-built bitstreams
proj Vivado project directory, with TCL scripts
src Interrupt Test FPGA Design HDL code
host Interrupt Test Host Program
bin Pre-built binaries (Windows only)
proj
linux Makefiles etc. for Linux
vxworks Makefiles etc. for VxWorks

```
|              |    ├─ win32vs2012 ·········· Microsoft Visual Studio 2012 projects (Windows)
|              |    └─ win32vs2013 ·········· Microsoft Visual Studio 2013 projects (Windows)
|              └─ src ························ Source code
├─ fpga
|    └─ repo
|         └─ vivado-2019.2 ·················· Repository for common Vivado IP
└─ host
     ├─ api-v1_4_21
     |    ├─ include ························ Header files for ADMXRC3 API
     |    └─ lib ···························· Libraries for ADMXRC3 API
     └─ app_framework-v1_4_0 ················ Example application framework
```

Figure 2 : Structure of this package

The root of this package, i.e. the directory which forms the root of the tree of directories and files making up this package, is referred to in the remainder of this document as **(ROOT)**.

The base directory of the FPGA design, i.e. **(ROOT)/example/itest-admxrcku1-v1_1_0** is referred to in the remainder of this document as **(DESIGN)**.

2 Design description

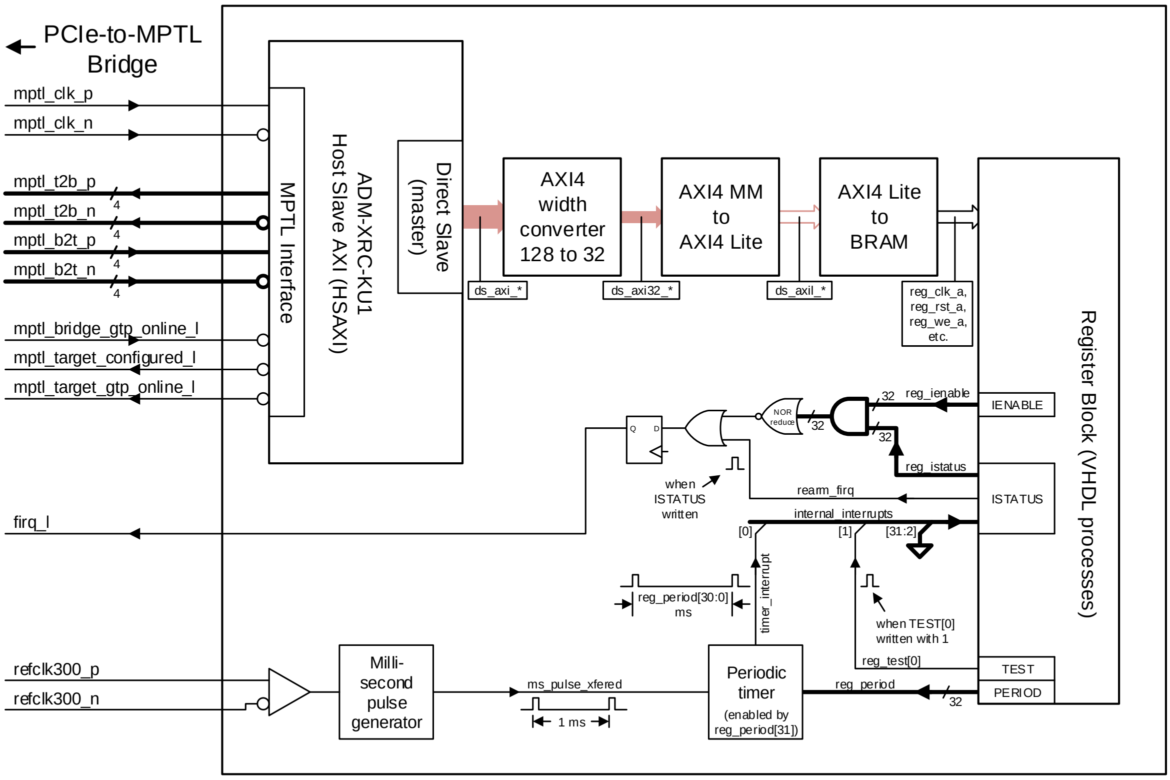


Figure 3 : Block diagram of Interrupt Test FPGA Design

The FPGA design consists of the following elements:

- An instance of **ADM-XRC-KU1-HSAXI** IP, supplied by Alpha Data, providing (amongst other things) an MPTL data channel and a memory mapped Direct Slave interface.
- Xilinx AXI infrastructure IP, for converting from 128-bit AXI4 memory-mapped interface of the **ADM-XRC-KU1-HSAXI** IP to 32-bit AXI4 Lite, for easier interfacing to registers.
- An AXI4 Lite to BlockRAM controller, used for interfacing the register block to AXI4 Lite.
- A register block, which can be read and written from the host via the **ADM-XRC-KU1-HSAXI** IP, implementing a simple interrupt controller and some other registers for generating interrupts.

The Direct Slave address map consists of a number of 32-bit wide registers:

Address	Semantics	Name	Purpose
0x00	R/W	IENABLE	If a particular bit is 1, the corresponding interrupt source is enabled. Reading returns the set of enabled interrupt sources.
0x04	R/W1S	IENABLE_SET	Writing a 1 to a particular sets the corresponding bit of IENABLE to 1. When read, returns the value of IENABLE.
0x08	R/W1C	IENABLE_CLR	Writing a 1 to a particular clears the corresponding bit of IENABLE to 0. When read, returns the value of IENABLE.

Table 1 : Direct Slave AXI4 address map (continued on next page)

Address	Semantics	Name	Purpose
0x0C	R/W1C	ISTATUS	Writing a 1 to a particular bit causes the corresponding interrupt source to become non-pending. Reading returns the set of pending interrupt sources.
0x10	WO	TEST	Writing a 1 to bit [0] sets interrupt source 0 to pending. Writing to bits [31:1] has no effect. Reading returns 0.
0x20	R/W	PERIOD	If bit [31] is 1, a periodic timer is enabled. When enabled, interrupt source 1 is periodically set to pending after an interval in milliseconds equal to bits [30:0].
Other			Reserved; must not be written, and reading returns undefined data.

Table 1 : Direct Slave AXI4 address map

3 Building the FPGA design

Tcl scripts to create the Vivado projects for the various configurations of the FPGA design are found in the **(DESIGN)/fpga/proj/** directory. These can be **source**d within the Vivado GUI, or **source**d by Vivado in batch mode. The available Tcl scripts are listed in Table 2:

Configuration	Project creation script in **(DESIGN)/fpga/proj/**
ADM-XRC-KU1 with KU060-2I	mkxpr-itest-ku060_2i.tcl
ADM-XRC-KU1 with KU115-2I	mkxpr-itest-ku115_2i.tcl

Table 2 : Project creation scripts by configuration

To generate a project, start a shell or command prompt, and issue a command of the following form:

```
cd (DESIGN)/fpga/proj
vivado -mode batch -source mkxpr-itest-ku060_2i.tcl
```

(Windows users should use backslashes in the **cd** command, rather than forward slashes.)

After the project has been created using the script, it can be opened in the Vivado GUI.

A TCL script is also provided in the same directory to fully rebuild the Vivado project via the shell or Command Prompt. This is named similarly to the **mkxpr** script, except that the prefix is **rebuild**. For example, to rebuild the Vivado project, invoke Vivado as follows:

```
cd (DESIGN)/fpga/proj
vivado -mode batch -source rebuild-itest-ku060_2i.tcl
```

(Windows users should use backslashes in the **cd** command, rather than forward slashes.)

Assuming that building is successful, the newly-built **.bit** file is:

(DESIGN)/fpga/proj/<project directory>/itest.runs/impl_1/itest.bit

Note
Pre-built bitstreams, which are found under the directory **(DESIGN)/fpga/bit/**, are **not** overwritten when the FPGA design is built.

4 Demonstration program

The demonstration program, **itest** is located in **(DESIGN)/host/src/** and consists of three source files:

- **cmdline.cpp**

  This file contains the **main** entry point and code for parsing command-line arguments, and nothing in this file is directly related to the FPGA design. It makes use of the **CExAppCmdLineArgs** class, which is provided by the example application framework code in **(ROOT)/host/app_framework-v1_4_0/**.

  Note that when building the demonstration program for VxWorks, this source file is omitted because a VxWorks downloadable kernel module (DKM) does not have a traditional main()-style entry point.

- **itest.cpp**

  This file contains code that actually drives the FPGA design and performs the interrupt test. It makes use of some classes for operating system abstraction (e.g. **CExAppThread**), also provided by the example application framework code.

- **itest.h**

  This file defines the interface to the code in **itest.cpp**, and is used by **cmdline.cpp**.

  Note that in VxWorks, the functions whose prototypes are defined in this file can be called from the VxWorks kernel shell.

The demonstration program works as follows:

1. It opens an ADMXRC3 device either by index or serial number, depending on arguments passed on the command line.
2. It maps the portion of Direct Slave space that covers the registers detailed in Table 1 into the process' address space, so that it can efficiently manipulate the registers in the FPGA.
3. It configures the target FPGA with the appropriate pre-built **.bit** file from the **(DESIGN)/fpga/bit/** directory.
4. It launches an interrupt-handling thread which is responsible for consuming notifications from the FPGA and taking action in response to them. In this example, the action taken is to merely display a message each time it consumes a notification. Each source of interrupt in the FPGA has a different message, however, demonstrating that the program can distinguish between multiple sources of interrupt in the FPGA.
5. The main thread configures the periodic timer to assert interrupt source 1 every 5 seconds, and enables interrupt sources 0 and 1 in the FPGA.
6. The main thread then enters a loop, waiting for the user to enter one of two commands via the **stdin** stream:
   - The "i" command causes the main thread to write a 1 to bit 0 of the **TEST** register, which causes interrupt source 0 to be set pending.
   - The "q" command causes the main thread to exit the loop.
7. The main thread causes the interrupt-handling thread to terminate.
8. Finally, the main thread cleans up and closes the ADMXRC3 device.

5 Building the demonstration program

5.1 Building in Linux

A **Makefile** for GNUMake is provided for building the demonstration program, **itest**. The GNU C++ toolchain and associated C and C++ development packages must be installed in the system that is used to build **itest**.

To build **itest**, follow this procedure:

1. Start a shell and change directory to **(DESIGN)/host/proj/linux**.
2. Issue the following command:

   ```
   make
   ```

Assuming that building is successful, the executable is **(DESIGN)/host/proj/linux/itest**.

The above procedure builds **itest** natively, i.e. for the architecture that the GNU toolchain on the build system targets by default. There are two variables that may be passed on the **make** command-line or set in the environment in order to change the way building is performed:

- **BIARCH**

  For most 64-bit Linux distributions, it is possible to build both a native (64-bit) executable and a 32-bit executable. To do this, set **BIARCH** variable to yes on the **make** command-line. For example:

  ```
  make BIARCH=yes
  ```

  Assuming that building is successful, the executables produced are **itest** (native 64-bit) and **itest32** (32-bit).

- **CROSS_COMPILE**

  To build using a cross-compiler, set the **CROSS_COMPILE** environment variable to the prefix of the toolchain binaries, ensuring that the toolchain is in the **PATH**. For example

  ```
  export PATH=/path/to/toolchain:$PATH
  export CROSS_COMPILE=arm-none-linux-gnueabi-
  make
  ```

- **SYSROOT**

  Generally used only when cross-compiling, the value of **SYSROOT** points to the target system's root filesystem. This may be required if the toolchain used for cross-compiling does not have the required defaults for paths to system header files and libraries directories. For example:

  ```
  export PATH=/path/to/toolchain:$PATH
  export CROSS_COMPILE=arm-none-linux-gnueabi-
  make SYSROOT=/path/to/arm-rootfs
  ```

5.2 Building in Windows

Solutions for Microsoft Visual Studio 2012 & 2013 are provided for building the demonstration program, **itest.exe**.

To build **itest.exe** for a particular configuration-platform combination, follow this procedure:

1. If using Microsoft Visual Studio 2012, open the solution **(DESIGN)/host/proj/win32vs2012/itest.sln**.

   If using Microsoft Visual Studio 2013, open the solution **(DESIGN)/host/proj/win32vs2013/itest.sln**.
2. From the **Standard** toolbar, which is visible by default in Microsoft Visual Studio, select the configuration and platform of interest; for example **Release**, **x64**.
3. From the main menu, select **BUILD** -> **Rebuild Solution**.

Alternatively, follow this procedure to build all available configuration-platform combinations of **itest.exe**:

1. If using Microsoft Visual Studio 2012, open the solution **(DESIGN)/host/proj/win32vs2012/itest.sln**.

   If using Microsoft Visual Studio 2013, open the solution **(DESIGN)/host/proj/win32vs2013/itest.sln**.

2. From the main menu, select **BUILD** -> **Batch Build...**
3. In the **Batch Build** dialog, click **Select All** and then **Rebuild**.

Once built, the executable files for **itest.exe** are located as follows, according to Visual Studio version, configuration and platform:

Visual Studio	Configuration	Platform	Executable location
2012	Debug	Win32	**(DESIGN)/host/proj/win32vs2012/itest/Debug/**
2012	Debug	x64	**(DESIGN)/host/proj/win32vs2012/itest/Debug64/**
2012	Release	Win32	**(DESIGN)/host/proj/win32vs2012/itest/Release/**
2012	Release	x64	**(DESIGN)/host/proj/win32vs2012/itest/Release64/**
2013	Debug	Win32	**(DESIGN)/host/proj/win32vs2013/itest/Debug/**
2013	Debug	x64	**(DESIGN)/host/proj/win32vs2013/itest/Debug64/**
2013	Release	Win32	**(DESIGN)/host/proj/win32vs2013/itest/Release/**
2013	Release	x64	**(DESIGN)/host/proj/win32vs2013/itest/Release64/**

Table 3 : Location of itest.exe

5.3 Building for VxWorks

A **Makefile** is provided for building a downloadable kernel module, **admxrc3ITest.out**, which has entry points that may be called from the VxWorks shell.

To invoke the **Makefile**, follow these steps:

1. Start a VxWorks Development Shell. This can be started from within Workbench or from the Start Menu if running in Windows.
2. In the shell, change directory to **(DESIGN)/host/proj/vxworks**.
3. Issue the **make** command, specifying the CPU architecture, toolchain and other build options. For example:

   ```
   make CPU=NEHALEM TOOL=icc VXBUILD="LP64 SMP" clean default
   ```

   The above command builds **admxrc3ITest.out** for 64-bit SMP Nehalem architecture using the Intel toolchain.

Assuming that the **make** command is successful, the build product is **admxrc3ITest.out**, which can be downloaded to the target system.

For a more detailed discussion of how to invoke the **Makefile**, refer to Appendix C.

6 Using the FPGA design

6.1 Using the FPGA design with a Linux host

The demonstration program, **itest**, runs on the host system's CPU and verifies that the FPGA design works as expected. Before running it, please ensure that your environment meets the following requirements:

- An ADM-XRC-KU1 is plugged into an XMC slot in the test machine and SW1-3 (Bridge Bypass mode) is OFF.
- ADB3 Driver 1.4.17 or later is installed in the test machine.
- You have built the demonstration program as detailed in Section 5.
- You are logged in as a user that is capable of executing programs as **root** using **sudo**.

Start the ADB3 Driver

If the ADB3 Driver is not already started, start it using the command:

```
sudo modprobe adb3
```

Run the demonstration program

To run the **itest** program with default arguments, issue the following commands in a shell:

```
cd (DESIGN)/host/proj/linux
sudo ./itest
```

This should (initially) yield output as follows:

```
INFO:  Configuring target FPGA with '../../../../fpga/bit/itest-ku060_2i/itest.bit'
...
INFO:  Creating interrupt handler thread ...
INFO:  Setting periodic timer to interrupt every 5 s ...
INFO:  [Interrupt Thread] Starting (Linux implementation).
INFO:  Enabling internal FPGA interrupts 0 & 1 ...
INFO:  Enter 'i' to generate FPGA internal interrupt 0, or enter 'q' to quit:
```

At this point, the user may either wait a few seconds for a periodic timer interrupt to occur, or enter the "i" command in order to generate an interrupt immediately. Continuing the above session, typical output (including commands entered by the user) looks like this:

```
i
INFO:  Writing 0x1 to TEST register ...
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 0 (caused by TEST register
       write).
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 1 (caused by periodic timer).
i
INFO:  Writing 0x1 to TEST register ...
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 0 (caused by TEST register
       write).
i
INFO:  Writing 0x1 to TEST register ...
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 0 (caused by TEST register
       write).
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 1 (caused by periodic timer).
q
INFO:  Terminating interrupt handler thread ...
INFO:  [Interrupt Thread] Terminating.
```

6.2 Using the FPGA design with a Windows host

The demonstration program, **itest**, runs on the host system's CPU and verifies that the FPGA design works as expected. Before doing so, please ensure that your environment meets the following requirements:

- An ADM-XRC-KU1 is plugged into an XMC slot in the test machine and SW1-3 (Bridge Bypass mode) is OFF.
- ADB3 Driver 1.4.17 or later is installed in the test machine.
- You are either:
  - Logged in as a user with Administrator privileges in a system without User Account Control (UAC) or where UAC is disabled, and have started a Windows Command Prompt (which will be elevated).
  - Logged in as a user with Administrator privileges in a system with User Account Control, and have started a Windows Command Prompt using "Run as administrator".

Run the demonstration program

To run the **itest.exe** program with default arguments, issue the following commands in the Windows Command Prompt:

```
cd (DESIGN)\host\bin\win32\x86
itest
```

This should yield output as follows:

```
INFO:  Configuring target FPGA with '..\..\..\..\fpga\bit\itest-ku060_2i\itest.bit'
...
INFO:  Creating interrupt handler thread ...
INFO:  Setting periodic timer to interrupt every 5 s ...
INFO:  [Interrupt Thread] Starting (Windows implementation).
INFO:  Enabling internal FPGA interrupts 0 & 1 ...
INFO:  Enter 'i' to generate FPGA internal interrupt 0, or enter 'q' to quit:
```

At this point, the user may either wait a few seconds for a periodic timer interrupt to occur, or enter the "i" command in order to generate an interrupt immediately. Continuing the above session, typical output (including commands entered by the user) looks like this:

```
i
INFO:  Writing 0x1 to TEST register ...
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 0 (caused by TEST register
       write).
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 1 (caused by periodic timer).
i
INFO:  Writing 0x1 to TEST register ...
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 0 (caused by TEST register
       write).
i
INFO:  Writing 0x1 to TEST register ...
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 0 (caused by TEST register
       write).
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 1 (caused by periodic timer).
q
INFO:  Terminating interrupt handler thread ...
INFO:  [Interrupt Thread] Terminating.
```

6.3 Using the FPGA design in VxWorks

The demonstration program, **admxrc3ITest.out**, runs on the VxWorks target machine and verifies that the FPGA design works as expected. Before running it, please ensure that your environment meets the following requirements:

- An ADM-XRC-KU1 is plugged into an XMC slot in the VxWorks target machine and SW1-3 (Bridge Bypass mode) is OFF.
- ADB3 Driver 1.4.17 or later has been built and is running on the VxWorks target machine. This can be done by one of two methods:
  (a) By downloading ADB3 Driver, as a set of downloadable kernel modules, to the VxWorks target machine, after booting. For this method, please refer to the release notes for **ADB3 Driver for VxWorks**.
  (b) By building ADB3 Driver Component into the VxWorks kernel so that it is automatically started when the kernel boots. For this method, please refer to the release notes for **ADB3 Driver Component for VxWorks**.
- You have built the demonstration program as detailed in Section 5.3.
- You have access to the kernel shell on the VxWorks target machine, either using a serial connection or using telnet.

Download the demonstration program to the VxWorks target machine

Assuming that you have built it as described in Section 5.3, the DKM for the demonstration program must first be downloaded to the VxWorks target machine. This can be done by a shell command such as:

```
-> ld <HOST:(DESIGN)/host/proj/vxworks/admxrc3ITest.out
value = -140737478303728 = 0xffff800000996010
```

where *HOST* is the VxWorks host.

Undefined symbols when loading the DKM
If the **ld** command fails due to undefined symbols, the most likely cause is that the ADB3 Driver has not been correctly downloaded to the VxWorks target system.

Run the demonstration program

Once the DKM for the demonstration program is resident in the VxWorks target system, it is possible to run it. The basic form of shell command that runs the program uses the **admxrc3ITestIndex** entry point in the DKM, and requires the path to the **(DESIGN)/fpga/bit/** directory to be the first argument:

```
-> admxrc3ITestIndex "HOST:(DESIGN)/fpga/bit",(int*)0,0
```

where *HOST* is the VxWorks host.

Successfully running the program as described above should yield output of the form:

```
INFO:  Configuring target FPGA with 'HOST:(DESIGN)/fpga/bit/itest-ku060_2i/itest.b
it' ...
INFO:  Creating interrupt handler thread ...
INFO:  Setting periodic timer to interrupt every 5 s ...
INFO:  [Interrupt Thread] Starting (VxWorks implementation).
INFO:  Enabling internal FPGA interrupts 0 & 1 ...
INFO:  Enter 'i' to generate FPGA internal interrupt 0, or enter 'q' to quit:
```

At this point, the user may either wait a few seconds for a periodic timer interrupt to occur, or enter the "i" command in order to generate an interrupt immediately. Continuing the above session, typical output (including commands entered by the user) looks like this:

```
i
INFO:  Writing 0x1 to TEST register ...
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 0 (caused by TEST register
       write).
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 1 (caused by periodic timer).
```

```
i
INFO:  Writing 0x1 to TEST register ...
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 0 (caused by TEST register
       write).
i
INFO:  Writing 0x1 to TEST register ...
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 0 (caused by TEST register
       write).
INFO:  [Interrupt Thread] Saw FPGA internal interrupt 1 (caused by periodic timer).
q
INFO:  Terminating interrupt handler thread ...
INFO:  [Interrupt Thread] Terminating.
```

Appendix A: Running the demonstration program in Linux & Windows

The demonstration program, **itest[.exe]**, may be invoked with a number of options:

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

Options begin with '-'. If an option requires a value, it may be specified in one or two forms: **-option=*<value>*** or **-option *<value>***. The available options are:

- **-h**, **-help**, **-?**

  This option displays a brief help message.

- **-index *<index>***

  This option specifies which reconfigurable computing device is to be used for the test. Zero corresponds to the first reconfigurable computing device in the system, as enumerated by the operating system. 1 corresponds to the second device, and so on.

  If omitted, the value is 0. This option cannot be specified along with the **-sn** option (see below).

  Examples:

  - **-index 0**

    Use the first reconfigurable computing device in the system.

  - **-index 10**

    Use the 11th reconfigurable computing device in the system.

  - **-index 0x2**

    Use the third reconfigurable computing device in the system.

- **-sn *<serial number>***

  This option specifies the serial number of the reconfigurable computing device that is to be used for the test.

  If omitted, the device used is chosen according to the **-index** option (see above). This option cannot be specified along with the **-index** option (see above).

  Examples:

  - **-sn 159**

    Use the reconfigurable computing device with serial number 159.

  - **-sn 0x5555**

    Use the reconfigurable computing device with serial number 0x5555 (21845).

Appendix B: Demonstration program entry points in VxWorks

The demonstration program can be invoked via two entry points in the **admxrc3ITest.out** DKM. These entry points are defined by the header file, **(DESIGN)/host/src/itest.h**, as follows:

```
int
admxrc3ITestIndex(
  const TCHAR* pBitPath,
  const TCHAR* pBitFile,
  unsigned int index);

int
admxrc3ITestSN(
  const TCHAR* pBitPath,
  const TCHAR* pBitFile,
  uint32_t     serialNumber);
```

Use of TCHAR

The demonstration program is portable between Linux, Windows and VxWorks. For this reason, **TCHAR** is used as the character data type, and when building for VxWorks, **TCHAR** is aliased to **char**.

- **admxrc3ITestIndex**

  This entry point is for running the demonstration program on an ADM-XRC-KU1 with a particular zero-based **index**. If there is only one card in the system, its index is always 0.

- **admxrc3ITestSN**

  This entry point is for running the demonstration program on an ADM-XRC-KU1 with a particular **serial number**.

The parameters are as follows:

- **pBitPath**

  If non-NULL, the **pBitPath** argument specifies the directory on the host filesystem where the pre-built **.bit** files are located. It is used as the prefix for constructing a full path to the **.bit** file to be used to configure the target FPGA, which is performed as follows (where + represents string concatenation):

  pBitPath + "/itest-**<device>**_**<speed><tempgrade>**[_**<step>**]/itest.bit"

  where "device", "speed", "tempgrade" and "step" are all obtained via the **ADMXRC3_GetFPGAInfo** function of the ADMXRC3 API. The "step" value is generally empty for a board fitted with a production silicon FPGA, and in that case is omitted from the **.bit** file path.

  For example, for an ADM-XRC-KU1 fitted with a KU115-2I device, the full path of the **.bit** file is constructed as:

  pBitPath + "/itest-ku115_2i/itest.bit"

- **pBitFile**

  If non-NULL, the **pBitFile** argument directly specifies the **.bit** file to use to configure the FPGA. Its value overrides whatever **.bit** file path was constructed from **pBitPath**.

  If **pBitPath** is NULL, **pBitFile** must be given a non-NULL value so that the program knows what **.bit** file to use.

- **index**

  In the **admxrc3ITestIndex** entry point, this parameter specifies the zero-based index of the reconfigurable computing card to use. If there is only one reconfigurable computing card in the system, its index is always zero. When there are more than one, indices are assigned by the system, generally according to the order in which they are discovered.

- **serialNumber**

  In the **admxrc3ITestSN** entry point, this parameter specifies the serial number of the reconfigurable computing card to use.

If **pBitFile** is not NULL, it overrides any value passed for **pBitPath**. Table 4 summarizes the interaction of **pBitPath** and **pBitFile**:

pBitPath	pBitFile	Behavior
NULL	NULL	Illegal; the program does not know what **.bit** file to use.
non-NULL	NULL	The program constructs the full path of the **.bit** file from **pBitPath** and information obtained via **ADMXRC3_GetFPGAInfo**.
N/A	non-NULL	The program uses **pBitFile** as the full path of the **.bit** file

Table 4 : Interaction of pBitPath and pBitFile

Appendix C: Makefile variables in VxWorks

The **Makefile** for building the downloadable kernel module **admxrc3ITest.out** in VxWorks can be invoked with a number of variables for controlling how the build is performed. The general form is:

```
make [CPU=<arch>] [TOOL=<tool>] [VXBUILD="[option] ..."] [target ...]
```

The available build targets for **make** are:

- **clean**

  This deletes all build products and intermediate files. When rebuilding with different values for **CPU**, **TOOL** etc. with respect to the previous build, first perform a clean.

- **default**

  This builds the product **admxrc3ITest.out** according to the values for **CPU**, **TOOL** etc.

To perform a full rebuild, use both **clean** and **default** together in the same command, in that order.

The variables that may be passed on the **make** command-line are:

- **CPU=*<arch>***

  Here, ***<arch>*** is the CPU architecture of the target system; for example **PPC604**, **NEHALEM**, **ARMARCH4** etc.

  If this variable is omitted, it defaults to **PPC604**.

- **TOOL=*<tool>***

  Here, ***<tool>*** is the toolchain that is to be used to build the DKM and, as of VxWorks 6.9, can be **diab**, **gnu** or **icc**.

  If this variable is omitted, it defaults to **gnu**.

- **VXBUILD="*[option]* ..."**

  Here the properties of the kernel of the target system must be specified. Including **LP64** means that the kernel of the target system is a 64-bit kernel. Including **SMP** means that the kernel of the target system is symmetric multiprocessing (SMP). Any options that are included should be separated by spaces, with all options together enclosed in quotes. For example, for a 64-bit SMP kernel, use

  ```
  VXBUILD="LP64 SMP"
  ```

  If this variable is omitted, it defaults to "", the result of which depends upon the defaults for the architecture selected by **CPU**. For example, if **CPU** is **PPC604** or **NEHALEM**, omitting **VXBUILD** results in building for a 32-bit uniprocessor kernel.

Hence, to fully rebuild for a 32-bit uniprocessor PowerPC 604 kernel using the GNU toolchain, issue the command

```
make clean default
```

To build for a 64-bit SMP Nehalem kernel using the Intel toolchain, issue the command

```
make CPU=NEHALEM TOOL=icc VXBUILD="LP64 SMP" default
```

Revision History

Date	Revision	Nature of change
7 Feb 2018	1.0	Initial version.
5 Sep 2022	1.1	Updated for Vivado 2018.3 to 2022.1.