Software Development Kit User Guide (UG107)

Speedster FPGAs

Preliminary Data

Achronix®
Data Acceleration

Copyrights, Trademarks and Disclaimers

Preliminary Data

This document contains preliminary information and is subject to change without notice. Information provided herein is based on internal engineering specifications and/or initial characterization data.

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com

Table of Contents

Chapter - 1: Introduction

The Achronix Software Development Kit (SDK) is a set of functions and data structures which enable users of the Speedster®7t FPGA family to write applications that communicate with and control their designs using the PCIe interface. The SDK consists of pre-compiled binary (private) libraries, source code for common public libraries, and source code for several example applications showing how common features can be implemented.

The SDK also contains the source code for a Linux PCIe device driver. The SDK can optionally be compiled to support either this native Achronix driver, or the PCIe driver from BittWare depending on user requirements. Contact BittWare to obtain the installer for their PCIe driver, if desired.

Chapter - 2: SDK Software Stack

The Achronix SDK is built on top of either the BittWare or Achronix device driver library. Either driver library supplies low-level routines that provide the following functions:

- Bind to a VectorPath® S7t-VG6 accelerator card
- Open and close the VectorPath device
- Perform memory-mapped reads and writes to the device
- Allocate DMA buffers
- Perform other hardware-specific tasks

The device driver itself is built on top of the Linux kernel API. In order to support both BittWare and Achronix drivers, the Achronix SDK includes a set of functions called driver translations. The Achronix driver translations act as a layer between the Achronix SDK code and the native driver functions, providing common functionality between the Achronix and BittWare drivers and allow for reuse of application code that supports both drivers.

Application code that must communicate with the PCIe device therefore operates by calling the Achronix SDK APIs and linking with the Achronix (and, optionally, BittWare) SDK shared libraries.

A conceptual diagram of the software stack is shown in the following figure.

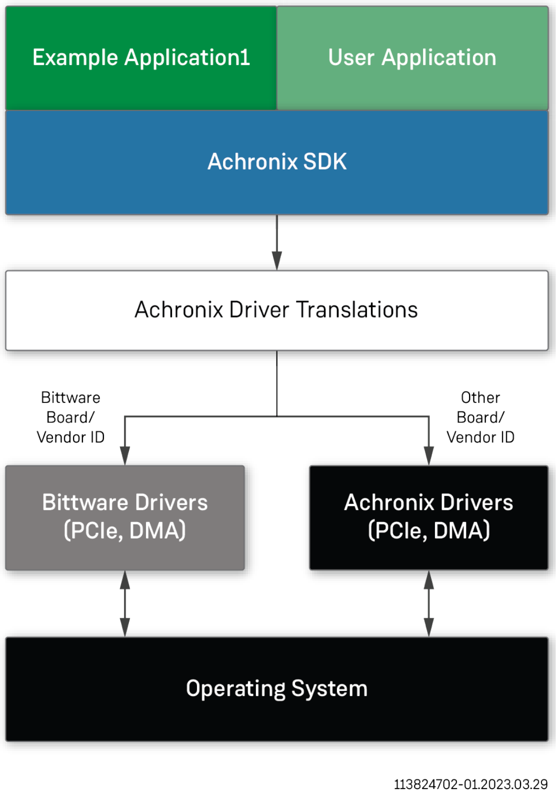
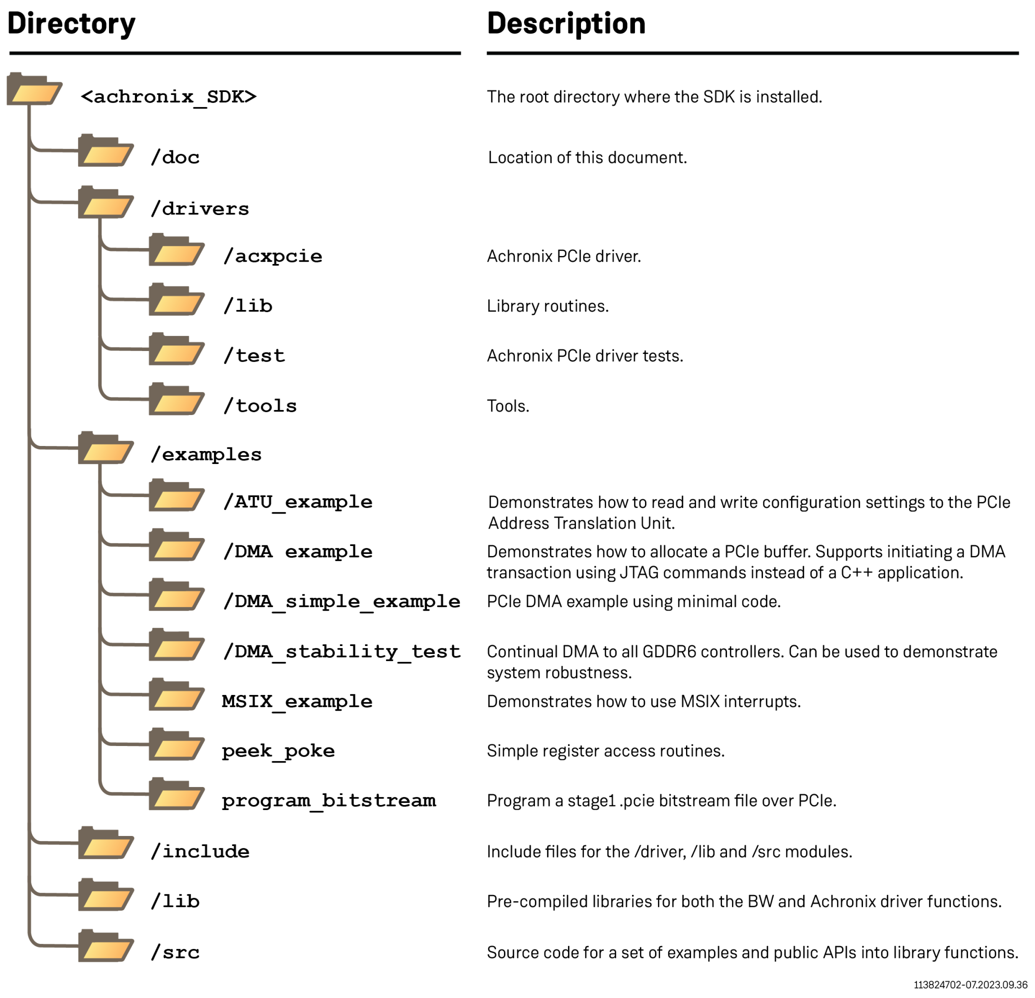

Example Application1
User Application
Achronix SDK
Achronix Driver Translations
Bittware Board/ Vendor ID
Other Board/ Vendor ID
Bittware Drivers (PCIe, DMA)
Achronix Drivers (PCIe, DMA)
Operating System
113824702-01.2023.03.29

Figure 1: *Software Stack*

Chapter - 3: Downloading, Compiling, and Installing the Achronix SDK

The Achronix SDK is shipped in the form of a Linux ZIP file. That file may be downloaded, unzipped, and compiled in any suitable location. The example applications can even be run out of that same download location if the environment is configured correctly, but this user guide assumes that, after compilation, the SDK is installed into its standard location in `/opt/achronix`. Installation requires administrator (root) privileges. See the included `README.txt` file for details.

The unzipped Achronix SDK has the following directory structure:



Directory	Description
`<achronix_SDK>`	The root directory where the SDK is installed.
`/doc`	Location of this document.
`/drivers`	
`/acxpcie`	Achronix PCIe driver.
`/lib`	Library routines.
`/test`	Achronix PCIe driver tests.
`/tools`	Tools.
`/examples`	
`/ATU_example`	Demonstrates how to read and write configuration settings to the PCIe Address Translation Unit.
`/DMA example`	Demonstrates how to allocate a PCIe buffer. Supports initiating a DMA transaction using JTAG commands instead of a C++ application.
`/DMA_simple_example`	PCIe DMA example using minimal code.
`/DMA_stability_test`	Continual DMA to all GDDR6 controllers. Can be used to demonstrate system robustness.
`MSIX_example`	Demonstrates how to use MSIX interrupts.
`peek_poke`	Simple register access routines.
`program_bitstream`	Program a stage1 .pcie bitstream file over PCIe.
`/include`	Include files for the /driver, /lib and /src modules.
`/lib`	Pre-compiled libraries for both the BW and Achronix driver functions.
`/src`	Source code for a set of examples and public APIs into library functions.

Figure 2: ***Achronix SDK Directory Structure***

Prerequisites

The Achronix SDK includes the source code for a native PCIe device driver, allowing it to be used in a standalone configuration with no dependencies. The Achronix SDK can also optionally be compiled with support for the PCIe device driver from BittWare. Before compiling with the BittWare driver, first install the BittWare software development kit (SDK) for the VectorPath accelerator card, available from the BittWare VectorPath developer portal at https://developer.bittware.com. See the Support Article for help getting started. Access to the BittWare developer portal and the SDK download is available only to verified purchasers of the VectorPath accelerator card. Please contact BittWare Support to obtain a developer account.

The following shows the default locations for installation of the BittWare SDK:

```
BittWare SDK FIle Locations

BittWare include files : /usr/share/bittware-sdk/include
BittWare library files : /usr/lib/x86_64-linux-gnu
```

If the BittWare SDK is installed to a different location, the appropriate entries in each of the Achronix makefiles must be updated.

The Linux GCC compiler must also be installed and available in the path. The SDK requires C++ 14 support, available in GCC versions 5.2 and later.

Note

The Achronix SDK is currently only available for the Linux platform with an installed Achronix VectorPath S7t-VG6 accelerator card. It has been tested, and is certified to compile and link on Ubuntu Linux 20.04 LTS and Ubuntu Linux 22.04 LTS. For support on other Linux distributions, Microsoft Windows, or non-VectorPath cards, please contact support@achronix.com.

Downloading the Achronix SDK

The Achronix SDK package, including this document, is shipped in the form of a zip file with a name that includes the release number and date, such as `achronix_sdk_v1.9.1_2023_05_23-03_18_40.zip`. The file can be downloaded from the Achronix support portal, using the knowledge base article, How do I Download the Achronix SDK?.

Unzipping the Achronix SDK

Unzip the SDK using the standard Linux `unzip` command as follows, substituting the current version number and release date in the filename:

```
$ unzip achronix_sdk_v1.9.1_2023_05_23-03_18_40.zip
```

Compiling and Installing the Achronix SDK

The Achronix SDK must be compiled and installed before it can be tested or used to build a custom application. Build instructions are slightly different depending on whether the Achronix or the BittWare PCIe device driver is used. For this reason, compilation instructions have been split into two separate sections:

- For use with the Achronix driver, please follow instructions in the Compiling and Installing the Achronix SDK for use with the Achronix Driver (see page 19) section
- For use with the BittWare driver, please follow instructions in the Compiling and Installing the Achronix SDK for use with the Bittware Driver (see page 21) section

Warning!

If both the Achronix and BittWare drivers are installed, they can conflict with each other. Both drivers should *NOT* be installed at the same time unless they are configured to bind to different Vendor/Device ID pairs. Refer to the Modifying the Device Driver section for information about configuring devices to which the driver can bind.

Compiling and Installing the Achronix SDK for Use With the Achronix Driver

Support for the Achronix PCIe device driver is built by default. To build the driver and SDK libraries, follow these steps:

```
$ cd SDK      -  Change to the source code directory
$ make clean -  Delete all object (.o), shared object (.so) libraries, and executables
$ make        -  Compile the device driver, SDK, device driver test/tools applications, and SDK
example applications
```

The Achronix SDK libraries, `lib/libacxsdk-priv-ac.so` and `lib/libacxsdk-pub.so`, comprise all of the functions defined in the header files in the `/include` directory. These functions are available to any application by linking the shared object files into any application build.

The device driver is built as a kernel plugin object file, `drivers/acxpcie/acxpcie.ko`. An associated driver API library file, `drivers/lib/libacxdev.so`, comprises all of the functions defined in the `drivers/lib/acxdev_api.h` header file. Applications built to support the Achronix PCIe driver must also link to this shared object file.

Compiling with Debug Symbols

In order to facilitate debugging, the SDK libraries and driver can optionally be compiled with debug symbols. Follow these steps:

```
$ cd SDK       -  Change to the source code directory
$ make clean   -  Delete all object (.o), shared object (.so) libraries, and executables
$ make DEBUG=y -  Compile with debugging symbols
```

Installing the SDK and Device Driver

This user guide assumes that the SDK and device driver are installed into a standard location, which is under `/opt/achronix`. Follow these steps:

Note

The installation procedure must be run with administrator (root) privileges.

```
$ cd SDK            -  Change to the source code directory
$ sudo make install -  Install the SDK
```

The installation location may be changed with an option to the make command as follows:

Note

This procedure is inteded for advanced users. Additional setup might be required to start the driver, link and run applications from a non-standard location.

```
$ cd SDK                                         -  Change to the source code directory
$ sudo make install INSTALL_ROOT='/my/location' -  Install the SDK to an alternate location
```

The installer prints verbose messages to the console indicating where the different components are being installed. For example:

1. API include files are installed in `/opt/achronix/include`.
2. API shared library files are installed in `/opt/achronix/lib`.
3. Symlinks to the shared libraries are created in a common location, which could be (depending on the Linux distribution used):
    - `/lib`
    - `/lib64`
    - `/usr/lib`
    - `/usr/lib64`

4. Achronix PCIe driver files:
   - The driver plugin is installed in `/opt/achronix/driver/acxpcie.ko`
   - The driver plugin is also installed in `/lib/modules/$(uname -r)/extra/acxpcie.ko`
   - The driver test and tool binaries are installed in `/opt/achronix/bin`

Uninstalling the SDK and Device Driver

To remove the SDK and device driver files from the installation locations, follow these steps:

```
$ cd SDK              -  Change to the source code directory
$ sudo make uninstall -  Uninstall the SDK
```

The un-installation location may be changed with an option to the make command as follows:

```
$ cd SDK                                    -  Change to the source code directory
$ sudo make uninstall INSTALL_ROOT='/my/location' -  Uninstall the SDK from an alternate location
```

Starting the Device Driver

If the device driver has been installed in the standard location under `/lib/modules`, as previously described, it can be started automatically by rebooting the computer. The kernel finds and starts the driver automatically. It can also be started manually without a reboot using the following command, run as root:

```
$ sudo modprobe acxpcie - Start the Achronix device driver
```

If the FPGA is already programmed with a bitstream that supports PCIe, the driver binds to the device immediately. Otherwise, the driver binds to the device after the FPGA is programmed.

Compiling and Installing the Achronix SDK for Use With the BittWare Driver

Note

For information on installing/uninstalling, starting/stopping, and configuring the BittWare driver, please follow the BittWare installation instructions.

Compiling the Achronix SDK

Support for the BittWare PCIe device driver is not built by default. To build the SDK libraries, follow these steps:

```
$ cd SDK                   -  Change to the source code directory
$ make clean               -  Delete all object (.o), shared object (.so) libraries, and
executables
$ make USE_DRIVER=bittware -  Compile the SDK and SDK example applications
```

The Achronix SDK libraries, `lib/libacxsdk-priv.so` and `lib/libacxsdk-pub.so`, comprise all of the functions defined in the header files in the `/include` directory. These functions are then available to any application by linking the shared object files into any application build.

Compiling with Debug Symbols

In order to facilitate debugging, the SDK libraries and driver can optionally be compiled with debug symbols. Follow these steps:

```
$ cd SDK                              -  Change to the source code directory
$ make clean                          -  Delete all object (.o), shared object (.so) libraries, and
executables
$ make USE_DRIVER=bittware DEBUG=y -  Compile with debugging symbols
```

Installing the SDK

This user guide assumes that the SDK is installed into a standard location under `/opt/achronix`. Follow these steps:

Note

The installation must be run with administrator (root) privileges.

```
$ cd SDK                                   -  Change to the source code directory
$ sudo make USE_DRIVER=bittware install -  Install the SDK
```

The installation location may be changed with an option to the make command as follows:

Note

This procedure is intended for advanced users. Additional setup might be required to link and run applications from a non-standard location.

```
$ cd SDK                                                            -  Change to the source code
directory
$ sudo make USE_DRIVER=bittware install INSTALL_ROOT='/my/location' -  Install the SDK to an
alternate location
```

The installer prints verbose messages to the console indicating where the different components are being installed. For example:

1. API include files are installed in `/opt/achronix/include`.
2. API shared library files are installed in `/opt/achronix/lib`.

3. Symlinks to the shared libraries are created in a common location, which could be (depending on the Linux distribution used)
   - `/lib`
   - `/lib64`
   - `/usr/lib`
   - `/usr/lib64`

Uninstalling the SDK

To remove the SDK files from the installation locations, follow these steps:

```
$ cd SDK              -  Change to the source code directory
$ sudo make uninstall -  Uninstall the SDK
```

The un-installation location may be changed with an option to the make command as follows:

```
$ cd SDK                                        -  Change to the source code directory
$ sudo make uninstall INSTALL_ROOT='/my/location' -  Uninstall the SDK from an alternate location
```

Starting the Device Driver

See the documentation that comes with the BittWare SDK for instructions on starting the BittWare PCIe device driver.

Testing the Achronix PCIe Device Driver

To verify that the VectorPath card and the Achronix PCIe device driver (if used) have been installed properly, run the driver unit tests as follows:

1. Make sure that the VectorPath card has been plugged into the system, and that is is programmed with a bitstream containing a PCIe configuration.
2. Start the Achronix driver as previously described:

   ```
   $ sudo modprobe acxpcie
   ```

3. Verify that the VectorPath card has been enumerated by the host, and that the driver has bound to it using the built-in Linux `lsmod` and `lspci` commands:

   ```
   $ lsmod | grep acx
   acxpcie                 16384  0
   $ lspci -d 1b59:
   65:00.0 Non-Essential Instrumentation [1300]: Device 1b59:0069 (rev 01)
   ```

4. Another option is to also look at the end of the kernel `kern.log` file to verify that the correct driver has bound to the card. For example:

```
$ sudo tail -f /var/log/kern.log
Mar  7 20:42:24 sjc-lab47 kernel: [  767.684873] ac7t15xxmodule loaded: ACXDEV_2_0_0217dev
Mar  7 20:42:24 sjc-lab47 kernel: [  767.685016] ac7t15xx 0000:65:00.0: acxdev_probe(vendor:
0x1b59 device:0x0069)
```

5. Run the following driver unit test commands:

```
$ sudo /opt/achronix/bin/test_open
$ sudo /opt/achronix/bin/test_dmaalloc
$ sudo /opt/achronix/bin/test_dbi
```

Testing the Achronix SDK

To verify that the SDK libraries and examples have been compiled and installed properly, run the DMA example as follows:

1. Ensure that the VectorPath card is plugged into the system, and that it is programmed with the `pcie_gddr6_ddr4_vp_demo` design bitstream, which can be downloaded separately from the Achronix Support website.
2. Run the following command to see the available command line options:

```
$ <achronix_SDK>/examples/DMA_example/dma_example --help
```

3. Run the following command to perform a small DMA test:

```
$ <achronix_SDK>/examples/DMA_example/dma_example -b 0x400000 -d H2D2H_SIM -e DDR4 -f random
```

The test performs the following steps:

1. Allocates two 4MB buffers on the host server.
2. Fills the first buffer with random data.
3. Transfers the contents of the buffer from the host to the device (the H2D direction) into the DDR4 memory space.
4. Transfers the same data back from device to the host (the D2H direction) into the second memory buffer.
5. Compares the two buffers to verify that the data made the round trip without errors.
6. Computes the achieved bandwidth in each direction.

Chapter - 4: Modifying the Device Driver

One aspect of the driver typically must be modified for every end application. Every PCIe device must be assigned a vendor ID and a device ID code, both of which are 4-digit hex (16-bit binary) numbers. The vendor ID and device ID in the bitstream are configured in the ACE IP designer. In order to bind to the device in use, the driver code must be configured to recognize the same vendor ID and device ID pair programmed into the device. By default, the driver code is configured to recognize the Achronix vendor ID, `0x1b59`, and a common device ID, `0x0069`, used in most of the Achronix demonstration designs. The following instructions should be applied if using custom vendor ID and/or device ID codes.

Modifying the Achronix Device Driver

The Achronix PCIe device driver is intended to serve as a reference implementation that can be extended for custom applications. Do the following to add support for custom ID codes:

1. Modify the file `drivers/acxpcie/pci.h` to add `#define` values as needed:

```
// Supported PCIe Vendor ID codes
#ifndef PCI_VENDOR_ID_ACHRONIX
#define PCI_VENDOR_ID_ACHRONIX  0x1b59
#endif

// Supported PCIe device ID codes
#define PCI_DEVICE_ID_ACHRONIX_TEST1  0x0069
#define PCI_DEVICE_ID_ACHRONIX_TEST2  0xface
```

2. Modify the file `drivers/acxpcie/pci.c` to add new #define values to the `pci_device_ids` array:

```
const struct pci_device_id pci_device_ids[] = {
  {PCI_DEVICE(PCI_VENDOR_ID_ACHRONIX, PCI_DEVICE_ID_ACHRONIX_TEST1)},
  {PCI_DEVICE(PCI_VENDOR_ID_ACHRONIX, PCI_DEVICE_ID_ACHRONIX_TEST2)},
  {0}
};
```

Modifying the BittWare Device Driver

Contact BittWare technical support for instructions to add support for custom Vendor and Device ID values.

Chapter - 5: Developing Applications

To develop custom applications with the SDK, it is recommended to follow the same format as one of the existing example applications. It is necessary to include the same BittWare and Achronix SDK header files, and to link with the same set of shared library files. Consult the included makefiles for more detail.

Minimum Requirements

Compilation

The following are the minimum requirements necessary to compile and link the Achronix SDK into custom software:

- Include the Achronix SDK header files in the `/opt/achronix/include` directory into C/C++ code
- Include the Achronix SDK shared object files in the `/opt/achronix/lib` directory when linking
- If using the BittWare driver, ensure that the BittWare library files are installed in `/usr/lib/x86_64-linux-gnu` and that they are also included when linking
- If the Achronix SDK shared-object (`.so`) library files have been installed somewhere other than a standard location (e.g., the default `/lib` location), the environment variable `LD_LIBRARY_PATH` must be set to include the location where those libraries can be found

Developing Applications Without High-Level SDK Code

Achronix suggests using the Achronix SDK high-level functions and constructs for ease of development. However, it is recognized that, in some cases, it might be desired to forgo usage of the high-level SDK functions and constructs to specially tune applications for specific needs. In this case, Achronix suggests using the driver translations as a foundation for the application development to enable support for the use of current or future drivers.

Chapter - 6: The PCIe Programming Model

PCIe devices such as the VectorPath card are memory-mapped I/O devices, meaning that they present their capabilities to the host CPU as one or more regions of memory that are mapped into the host memory space. Software that reads from and writes to registers or memories on the FPGA implement those operations by reading or writing to the appropriate address on the host. Reads and writes between host memory and device memory, in either direction, can also be implemented using direct memory access (DMA) transactions though a DMA controller built into the Speedster7t AC7t1500 FPGA. Finally, the device can also signal the host using interrupts through the PCIe standard MSI-X protocol. Each of these topics are discussed in more detail in the following text.

In order to determine the correct address for a memory space in the FPGA, whether within the interface subsystems (such as GDDR6) or the fabric (such as a NAP), it is necessary to understand the differences between how the FPGA and the host software map their address spaces.

Linux Host Memory Mapped Addressing

In order to implement the memory-mapped I/O abstraction, a PCIe controller reads base address registers (BARs) in the device. The BARs specify mappings from the host machine physical address space onto the PCIe device internal address space(s). Each BAR defines the size of an address space, and the local (host) base address of a block of mapped memory. During PCIe enumeration, for each BAR, the host allocates a memory region of the size requested by the BAR, maps that memory into the host (usually 64-bit) physical memory space, and writes the base address of that memory region into the BAR.

On the device side, each BAR maps to a memory region of the same size in the FPGA address space. When host software must read from, or write to, a physical address covered by one of the BARs, it can request that the device driver map that BAR physical addresses into the application virtual address space. Reads from or writes to those virtual addresses are then intercepted by the host PCIe controller, and then routed to the correct location on the FPGA.

FPGA Memory Addressing

The FPGA maps all addresses into a 42-bit linear address space defined by the built-in 2D Network-on-Chip (2D NoC), giving access to all of the configuration/status registers, all external memory IP locations, and all internal network access points (NAPs) of the 2D NoC. In this 42-bit linear space, each interface subsystem can be addressed as a hierarchy of addresses, sub-divided as follows:

- Space – major different memory areas such as `CSR_SPACE` (interface subsystem registers), `NAP_SPACE` (NAPs in the programmable fabric) and `DDR4_SPACE`, `GDDR6_SPACE` (external memories).
- Target – either interface subsystem IP blocks normally within the `CSR_SPACE`, e.g., `PCIE_0`, `PCIE_1`, `ETHERNET_0`, etc., or individual external memory controllers, e.g., `GDDR6_0`, `GDDR6_1`, etc.
- IP ID – within an interface subsystem, individual blocks. So, for example, with Ethernet, the `CORE` registers, then `SERDES_0` and `SERDES_1` register areas.
- Address – the memory address.

The overall size within each of these areas varies. However, the two most common to be accessed by external software are `CSR_SPACE` and `NAP_SPACE`. They have the following addressing:

Table 1: ***CSR_SPACE Addressing***

Name	Space	Target	IP_ID	Register Address
CSR_SPACE	Bits [41:34]	Bits [33:28]	Bits [27:24]	Bits [23:0]

Table 2: ***NAP_SPACE Addressing***

Name	Space	NAP Column	NAP Row	Register Address
NAP_SPACE(1) (2)	Bits [41:35]	Bits [34:31]	Bits [30:28]	Bits [27:0]

Table Notes

1. The NAPs are numbered from 1 for placement constraints within the device. However, they are addressed from 0 in the above addressing table.
2. The SDK function `util_calc_nap_absolute_address()` returns bits [41:28] of the device address when given the NAP column and row locations (numbered from 1).

The 2D NoC and its address mappings are described in much greater detail in the *Speedster7t Network on Chip User Guide* (UG089).

Each physical function in the PCIe controller on the Speedster7t FPGA contains six 32-bit (or three 64-bit) BAR registers, each of which supports up to a 64MB address space. Because the Speedster 7t FPGA 2D Network-on-Chip (2D NoC) exposes a much larger 42-bit address space, it should be clear that the entire device address space cannot be mapped into the host in its entirety using a static BAR mapping. It is therefore necessary to be aware of those size limitations when designing applications, and to configure the BARs and their sizes to match those requirements.

The size of each BAR, and FPGA base address(es) assigned to each BAR, are defined for each physical function in the PCIe IP configuration menu in the ACE I/O designer, and stored in a hardware controller called the address translation unit (ATU). Refer to the *Speedster7t PCIe User Guide* (UG098), available under NDA, for more information about configuring ATU BAR mappings in the bitstream (see the knowledge base article, How do I gain Access to Confidential Documents?). However, the ATU configuration can also be modified by applications at runtime using functions in the SDK, providing one mechanism to overcome the mismatch between the maximum BAR sizes, and the size of the FPGA address space. The function and capabilities of the ATU are described in more detail in the section, Address Translation Unit (see page 31).

It is important to emphasize that BAR register configuration is specific to a design. The capabilities of a user application must match the BAR register sizes and device address assignments in the design. When mapping a BAR to a CSR register space, or the address of a NAP for example, the design and user application must agree on those BAR sizes and address mappings.

Warning!

The BAR register configurations are specific to a design. The user application must be written such that it aligns with those configurations.

Another method to overcome the mismatch between the maximum BAR sizes, and the size of the FPGA 42-bit address space, as well as to achieve much higher bandwidth, direct memory access (DMA) can be used instead of memory-mapped BAR reads and writes. Refer to the DMA Transfers (see page 33) section for more information about DMA.

Note

DMA accesses do not use BAR registers for the transfers. DMA transfers occur directly between the device addresses and the physical address within the host. BAR registers are used, however, for control and monitoring of the DMA transfers.

PCIe Configuration Example

To understand the requirement for alignment between the BAR assignments and ATU mappings in a design, and the implementation of applications written to run on that design, the demonstration design `pcie_gddr6_ddr4_vp_demo` is used as an example. That design demonstrates PCIe DMA to and from external memory, (DDR4 and GDDR6), and also to a NAP in the fabric, using the `DMA_example` application provided within the Achronix SDK.

Example BAR Configuration

PCIe IP configurations, including the BAR assignments, are specified in the ACE I/O designer as well. The I/O designer is accessed from the IP configuration perspective in the ACE GUI. The PCIe configuration parameters are written by the I/O designer into the design file `/src/acxip/pcie_express_x16.acxip`, which must be included in the ACE project specification. The BARs for this demonstration design are configured in the I/O designer as shown in the following table.

Table 3: ***Example BAR Mappings***

BAR	Type	Size (Bytes)	42-Bit 2D NoC Address	FPGA memory space
0	Memory	64M	`0x042_4000_0000`	NAP located in column 5, row 5.
1	Memory	1M	`0x043_e000_0000`	NAP located in column 8, row 7.
2	Interrupt	1M	`0x000_0000_0000`	Used by PCIe core for MSI-X interrupts.
3	Memory	1M	`0x081_9100_0000`	Base of `CSR_SPACEPCIE_1BASE_IP`.
4	Memory	1M	`0x002_0000_0000`	GDDR6 controller 0, channel 1.
5	Memory	1M	`0x100_0000_0000`	Base of DDR4 memory.

The two mandatory BAR mappings in this design are BAR 0 and BAR 3.

BAR 0 maps to a NAP located in column 5, row 5. Referencing the file `/src/constraints/ace_placements.pdc`, this NAP connects to a fabric-based register control block macro. This block provides status and control signals from/to the design, operating the DDR4 training transactions, and monitoring the GDDR6 training status among other functions.

```
Excerpt From ace_placements.pdc

# Register control block NAP placed in 5,5
set_placement -fixed {i:i_reg_control_block.i_axi_master.i_axi_master} {s:x_core.NOC[5][5].logic.
noc.nap_m}
```

BAR 1 maps to a NAP located in column 8, row 7. This NAP is connected to a BRAM responder macro, which makes a BRAM directly available to the host over PCIe and the 2D NoC.

BAR 2 provides access to the MSI-X vector tables and pending bits. It is used by the `MSIX_example` application.

BAR 3 is mapped to the control and status register (CSR) space within the PCIe core. This mapping is required to allow the software to access the registers that configure and control the DMA transfers. Without this mapping, the application would not be able to perform DMA.

Note

It is strongly recommended that all designs have one BAR mapped to the PCIe config status register (CSR) space (address `0x081_9100_0000`). A CSR BAR is required in the application to access any DMA or PCIe core functions.

BAR 4 and BAR 5 map the the base of the GDDR6 and DDR4 off-chip memory arrays, respectively.

Example Software Implementation

Referencing the `DMA_example` in `/demo/sw/examples/DMA_example/DMA_example.cpp`, it can be seen how BAR 0 and BAR 3 are used to access the register control block and the PCIe CSR registers, respectively.

```
Excerpt From DMA_example.cpp

// Configure mapping for each BAR.
// IMPORTANT - These can change on a per-design basis.
ACX_BAR_handle* reg_ctrl_bar = device.get_bar_handle(0);  // Mapped to the register control block
in the fabric
ACX_BAR_handle* csr_bar      = device.get_bar_handle(3);  // Mapped to the Control Status
Registers

....

// DDR4 training is controlled and monitored by the reg_ctrl block, which is
// accessed by one of the BARs
if ( acx::ddr4_run_training( device, reg_ctrl_bar, /*train_override*/ false, /*verbose*/ true) !=
0 ) {

....

// Initialize the DMA core. Resets both engines (read and write) and sets arbitration weights.
Only need to do this once.
acx::dma_init( device, csr_bar, part, pcie_core );
```

Implementation Recommendations

To ensure alignment between the device and the software, the following is recommended:

- Assign a BAR to the PCIe config status registers (offset `0x081_9100_0000`). Use this BAR for all PCIe CSR accesses in the software.
- If the design contains a register control block with a NAP, define a BAR with an address that matches the placement location of that NAP (defined in the `ace_placements.pdc` file). Use this BAR in the software for all accesses to registers in the register control block.
- DMA transfers do not use BARs. To obtain the device address of a NAP for DMA access, call `acx::util_calc_nap_absolute_address()`. For GDDR6 or DDR4 device addresses use the `<target>_SPACE` define from `/include/Achronix_SDK.h`, and add in the required memory offset.
- To perform memory-mapped reads and writes to device registers or memories without using DMA (sometimes called BAR reads and writes), define a BAR with a base address and size that covers the region of memory to be accessed. For example, refer to BAR 1, 4, and 5 defined in the Example BAR Mappings (see page 29) table. They map to a BRAM, GDDR6, and DDR4 memories, respectively.

Address Translation Unit (ATU)

Mappings between the host 64-bit physical address space, and the 42-bit device address space, are performed by the address translation unit (ATU), which is a part of the PCIe interface subsystem. When a user application reads from or writes to a physical address that has been assigned by the host OS to one of the device BARs, that raw address is passed to the ATU for translation from a host address to the device 2D NoC address. If the ATU were to be unconfigured, the host addresses would be truncated or zero-padded to 42-bits (depending on whether the design uses 64-bit or 32-bit BARs), and then passed directly on into the 2D NoC without translation. This could create an invalid address which could then cause 2D NoC access failures. Therefore, the ATU must be configured, using the ACE I/O designer, whenever a PCIe block is configured to include one or more BARs.

The ATU consists of 100 regions, each of which can be individually configured with a mapping between one contiguous region of host memory addresses, and a corresponding region of addresses (of the same size) on the device.

Warning!

If ATU regions are defined to be overlapping (either in the device code or the host side), the behavior is undefined.

Each ATU region can be in one of two modes, BAR match mode or address match mode, as illustrated in the following diagrams.

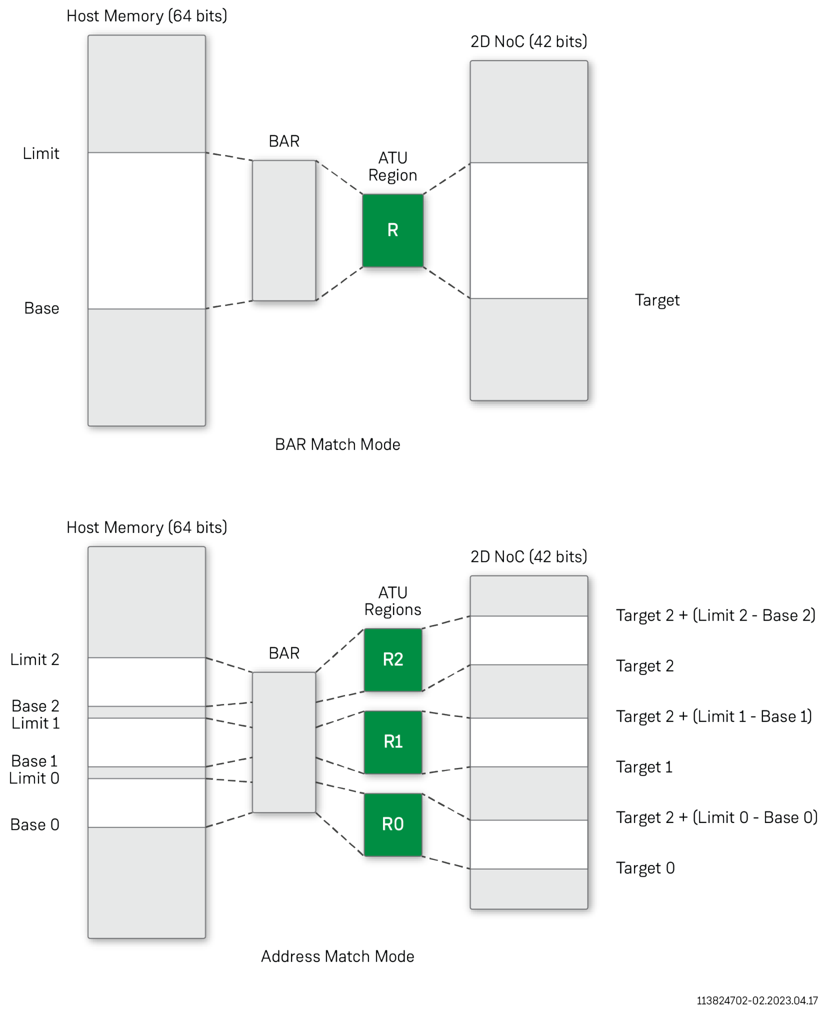


Figure 3: *ATU Region Examples*

BAR Match Mode

In BAR match mode, the ATU translates all of the addresses covered by one BAR (either 32-bit or 64-bit BARs) into a single block of device memory of the same size. Therefore, since each PCIe physical function contains only six 32-bit BARs, and since the PCIe controller supports up to four physical functions, a maximum of 24 ATU regions can be configured in BAR match mode. In this mode, only the device-side target address for the BAR must be specified, as the host-side base and limit addresses are assigned by the host operating system during enumeration.

Address Match Mode

Address match mode supports the full 100 ATU regions. Each region specifies base addresses in host memory and in device memory along with a region size. This scheme allows the addresses covered by a single BAR to be split up into a large number of individual slices that each map to any region of the 42-bit 2D NoC address space. For example, a single BAR could be made to cover 80 different BRAMs, each connected to its own NAP in the Speedster7t FPGA 2D NoC. It is necessary to ensure that the regions are non-overlapping.

MSI-X Interrupts

A device can generate interrupts on the host over PCIe using a mechanism called message signaled interrupts-extended (MSI-X). Using the ACE IP designer tool, a device can be configured with up to eight independent interrupt vectors. After the device signals an interrupt, the host kernel executes an interrupt handler, which can then trigger application code to service the interrupt. Interrupts can be masked, either asserting a global per-function mask, or an individual per-vector mask. If an interrupt is triggered while masked, the PCIe device asserts a pending bit, and defers the interrupt until the mask bit is de-asserted, at which point the PCIe device also de-asserts the corresponding pending bit.

The Achronix SDK provides an API for MSI-X interrupts. Various aspects of the MSI-X register context, including the pending bits, can be read and/or printed to the console. An interrupt can be triggered from the host for testing and debug purposes. The function and vector mask flags can be asserted and de-asserted. Finally, a function `msix_wait_for_interrupt()` can be called that returns when an interrupt has been received by the host.

DMA Transfers

Direct memory access (DMA) transfers do not require the use of BAR registers (other than for configuring the DMA engine itself), and are therefore not limited by the number of BARs or the BAR sizes. The DMA engine built into the PCIe controller can efficiently transfer blocks of data of any size directly between the host remote 64-bit address space and the local device 42-bit address space. The terms "local" and "remote" are from the perspective of the on-device DMA controller. Because that perspective can be confusing, transfers are referred to as host-to-device (H2D) or device-to-host (D2H) transfers, rather than reads and writes.

To perform a DMA transfer, the software must allocate a buffer on the host to source or receive the data, then configure DMA controller registers with the source and destination addresses and the number of bytes. When reading or writing the buffer from software, the buffer must be referred to using its virtual address. However, when programming the DMA controller with the host buffer source or destination address, the buffer physical address must be used. The `acxsdk::DmaHostBuffer` object makes it convenient to obtain both the virtual and physical addresses for the buffer.

Note

It is not necessary to have a BAR register map the memory location in the device for a DMA transfer.

2D NoC Physical Address Calculations

To obtain the 42-bit 2D NoC device physical address for a DMA transfer, the following methods are available:

- For a NAP in the fabric, call `acx::util_calc_nap_absolute_address()` with the NAP row and column index
- For DDR4, use the define `DDR4_SPACE` to provide the base address, and add any necessary offset
- For GDDR6, use the define `GDDR6_SPACE` to provide the base address and add any necessary offset

Note

Each of the 16 GDDR6 channels is addressed with `addr[36:33]`.

The following DMA example shows the use of these functions and defines for determining the device physical (local) address for a DMA transfer:

DMA Example

```
// Calculate the DMA target address in the device
uint64_t device_phys_base_addr = 0x0;
if (options.endpoint == acx::DDR4) {
    device_phys_base_addr = DDR4_SPACE
} else if (options.endpoint == acx::GDDR6) {
    device_phys_base_addr = GDDR6_SPACE;
} else if (options.endpoint == acx::NAP) {
    // AXI BRAM responder NAP location is set in project pdc file
    device_phys_base_addr = util_calc_nap_absolute_addr(part, axi_bram_resp_col,
axi_bram_resp_row);
}
```

Basic DMA Operation

In order to perform a DMA transfer, the software must first open and obtain a handle to the PCIe device. The Achronix SDK provides a C++ class named `acxsdk::PCIDevice`. The device is opened by calling the class constructor with the PCIe `device_id`. By default, `device_id` equals zero if there is only a single VectorPath card installed. In the event that multiple VectorPath cards are installed, the `device_id` is 0, 1, 2, etc., in the order that the devices were probed by the kernel. After the device is opened, the `PCIDevice` class is used as the device handle. The device is automatically closed in the class destructor when it goes out of scope or the application exits.

For DMA, the software is required to allocate a host buffer and, if performing a host-to-device DMA, copy the required data into the host buffer. The Achronix SDK utilizes the `acx_dma_malloc()` function to allocate that buffer, through a C++ class named `acxsdk::DMAHostBuffer`. The buffer is created by calling the class constructor with a pointer to the `PCIDevice` and the size of the buffer in bytes. The buffer is automatically deallocated when the `DMAHostBuffer` object goes out of scope, or when the application exits.

Note

Maximum buffer size is currently limited to 4MB and the device-side starting address must be 4-byte aligned.

Warning!

It is not possible to allocate the buffer with a simple `malloc()` call. The reason being the `DMAHostBuffer::get_phys_addr()` function is required to obtain the buffer physical (not virtual) address.

The following example illustrates using the `PCIDevice` and `DMAHostBuffer` constructors. The function `buffer.fill_random()` is used to fill the buffer with random data. The source for `fill_random()` is contained within `/src/Achronix_PCI.cpp`. This function can serve as an example how to fill the buffer with application-specific data using the buffer virtual address. The source code for `PCIDevice` and `DMAHostBuffer` are also both available in the `/src/Achronix_PCI.cpp` file. They can be customized as required.

```
acxsdk::PCIDevice device(options.device_id);
acxsdk::DMAHostBuffer buffer(&device, buffer_size_in_bytes);
buffer.fill_random();
```

1. Having opened the PCIe device and writing the required data into the `DMAHostBuffer`, the software must call `acxsdk::dma_init()` to initialize the DMA controller. This initialization only needs to be called once at the start of the application as long as only a single process is using the DMA engine. The function requires the following:
   - The `ACX_PCIE_dev_handle` pointer obtained by calling `acxsdk::DMAHostBuffer::get_device()`
   - The `ACX_BAR_handle` object for a BAR that maps to the PCIe CSR memory area in the FPGA, giving access to the DBI and PCIe core control registers
   - Defines for the desired Achronix part name and PCIe controller number

   ```
   acxsdk::dma_init(device.get_device(), csr_bar, acxsdk::AC7t1500ES0, acxsdk::PCIE_1);
   ```

2. For each individual DMA transaction, an instance of a `acxsdk::DmaCommand` struct must be populated with parameters that describe the transaction. The most important parameters are:
   - The transfer direction
   - The 42-bit `device_address` (calculated in the previous code example)
   - The 64-bit `host_address`, obtained by calling `DMAHostBuffer::get_phys_addr()`
   - The buffer size in bytes, obtained from the `DMAHostBuffer` class

   ```
   acxsdk::DmaCommand myDmaCommand;
   myDmaCommand.csr_bar        = csr_bar;
   myDmaCommand.pcie_core      = acxsdk::PCIE_1;
   myDmaCommand.dma_direction  = acxsdk::HOST_TO_DEVICE;
   myDmaCommand.dma_channel    = options.channel;
   myDmaCommand.device_address = device_phys_base_addr;
   myDmaCommand.host_address   = buffer->get_phys_addr();
   myDmaCommand.size_in_bytes  = buffer->get_size_in_bytes();
   myDmaCommand.descriptor_list_address = 0x0;
   ```

3. For each transfer, the DMA engine is configured with `acxdsk::dma_config()`, then the transfer started with `acxsdk::dma_start()`.

4. To wait for a DMA to complete, call the function `acxsdk::dma_wait()`. This function returns after the DMA has completed. In the event that the DMA transaction does not complete correctly, or times-out, the function `acxsdk::dma_halt()` must be called to abort the transaction before starting a new DMA transfer.

```
acxsdk::dma_config( device.get_device(), myDmaCommand );
acxsdk::dma_start(device.get_device(), myDmaCommand);
acxsdk::DmaStatus status = acxsdk::dma_wait(device.get_device(), myDmaCommand,
/*timeout_in_seconds*/2);
if (status == acxsdk::DMA_RUNNING) {
    acxsdk::dma_halt(device.get_device(), myDmaCommand);
    // code to recover and re-issue the command
}
```

Linked List Mode

In addition to the basic DMA operation just described, the more advanced linked list mode is available to handle larger DMA transfers, or for implementing more efficient streaming applications. In this mode, the DMA context (source address, target address, buffer size) is loaded into a data structure called a DMA descriptor, instead of being passed directly to `dma_config()` through the `DmaCommand` struct. Multiple buffers can be transferred in a single call to `dma_start()` by creating a descriptor for each buffer, and then combining the descriptors into a linked list. The descriptor list is then transferred into device memory, and the physical address of the list is passed into the `dma_config()` call through the `DmaCommand` struct. The descriptor list may be placed anywhere in device memory (DDR4, GDDR6, or a BRAM connected to a NAP). The descriptor list may be transferred into device memory using individual BAR writes, or (recommended) a small DMA transaction.

To aid in constructing a DMA descriptor list, the Achronix SDK provides a small class named `acxsdk::DMADescriptorList`. After creating the `DMADescriptorList`, the descriptor data is populated by calling the `acxsdk::build_data_descriptor()` function once for each `DMAHostBuffer`. In the following example, an array of host buffers, all of the same size have been allocated. The source code for the `DMADescriptorList` class is available in `/src/Achronix_PCI.cpp` and can be customized if required to suit the application.

```
acxsdk::DMADescriptorList descriptors(&device, options.num_descriptors, GDDR6_SPACE);
for (uint64_t i = 0; i < options.num_descriptors; i++) {
    acxsdk::dma_build_data_descriptor(descriptors[i],
        options.buffer_size_in_bytes,
        buffer_vec[i]->get_phys_addr(),
        device_phys_base_addr + (options.buffer_size_in_bytes * (uint64_t)i));
}
```

Every block of descriptors in a `DMADescriptorList` consists of one or more `DMADataDescriptors` and ends with a single `DMALinkDescriptor` that might link to another `DMADescriptorList`. The use of multiple linked `DMADescriptorLists` is beyond the scope of this document. The `DMADescriptorList` constructor populates the terminating `DMALinkDescriptor` with a pointer back to the first `DMADataDescriptor` in the list, which is the default configuration for a single unlinked descriptor list.

Note

The function `dma_build_link_descriptor()` is available to populate the `DMALinkDescriptor`.

1. After building the `DMADescriptorList`, the list must be transferred into device memory, which can be performed using the Basic DMA Operation (see page 34) procedure. For convenience, the `DMADescriptorClass` makes available the `get_device_phys_addr()` function which returns an address that is then passed to the class constructor. In the previous example, the physical address of token `GDDR6_SPACE` is used. This address equates to the lowest of GDDR6 memory addresses in the 42-bit 2D NoC address space. For more information, see the `DMA_example` source code.
2. After the descriptor list is complete, the `DmaCommand` structure is populated with the DMA transfer parameters. Comparing the following linked list mode example with the previous Basic DMA Operation (see page 34) example, it can been seen that the physical address of the descriptor list in device memory is used in place of the `device_address`, `host_address`, and `size_in_bytes` elements.

```
acxsdk::DmaCommand myDmaCommand;
myDmaCommand.csr_bar        = csr_bar;
myDmaCommand.pcie_core      = pcie_core;
myDmaCommand.dma_direction  = acxsdk::HOST_TO_DEVICE;
myDmaCommand.dma_channel    = options.channel;
myDmaCommand.verbosity      = options.verbosity;
myDmaCommand.device_address = 0x0;
myDmaCommand.host_address   = 0x0;
myDmaCommand.size_in_bytes  = 0x0;
myDmaCommand.descriptor_list_address = descriptors->get_device_phys_addr();
```

1. To initiate, start and wait the the DMA, the same commands, `dma_config()`, `dma_start()`, and `dma_wait()`, are used. The DMA controller performs all of the transfers specified in the descriptor list before returning from the `dma_wait()` function.

   Warning!

   DMA descriptor lists must be uploaded to the device and must exist in device memory during the DMA transfer. Care should be taken to prevent the DMA descriptor list and DMA buffer target device addresses from overlapping. If addresses do overlap, undefined behavior results during the transfer.

Chapter - 7: Design Requirements

This section documents the minimum requirements for designs that use various components of the SDK. See the referenced demo design for more information and an example of the following.

Achronix_DDR4.cpp

The DDR4 functions manage training of the DDR4 controller (using the `ddr4_training_polling_block` in the fabric) and, in addition, control sending and reception of data along with performance monitoring of the throughput to and from the DDR4 (using the `axi_pkt_gen`, `axi_pkt_chk` and `axi_performance_monitor` blocks in the fabric). In order to use these functions, the fabric must contain the preceding instances. The header file `/include/Achronix_DDR4.h` specifies the register control block addresses for the various DDR4 control blocks. These addresses should be modified to match the fabric design.

Achronix_GDDR6.cpp

The GDDR6 functions read the status of the Achronix device manager (ADM) that is configured to perform GDDR6 training. In order to use these functions, the fabric must contain an instance of the ADM configured to train at least one GDDR6 controller. The header file, `/include/Achronix_GDDR6.h`, specifies the register control block addresses for the ADM. These addresses should be modified to match the fabric design.

Achronix_PCI.cpp

The PCIe functions require at least one PCIe core to be enabled and configured within the device. Normally (on a VectorPath card) this is `PCIE_1` which connects to the primarily PCIe connector. In addition, the functions require one BAR that maps to the PCIe core registers. If DMA transfers are required to GDDR6 or DDR4, the appropriate training blocks for these interfaces must be instantiated within the fabric and suitable control must be available to ensure that the interfaces are correctly initialized and ready for read and write operations before any DMA or PCIe BAR access is made. See the previous functions for control and monitoring of these external memories.

If DMA descriptors are required to be stored in a BRAM attached to a NAP, for internal fast storage, an `axi_bram_responder` instance is required in the fabric.

DMA_example.cpp

The DMA example code has the same requirements as `Achronix_PCI.cpp` in that a single core must be present and configured, and that any external memory interfaces have been correctly initialized before use.

Chapter - 8: SDK Functions

The SDK library includes the following functions. Function prototypes are defined in `<achronix_SDK>/include /Achronix_PCI.h`.

Quick Reference Table

A list of all current functions, with their arguments follows.

```
// General utility functions
uint64_t util_calc_nap_absolute_addr (PartName part, int col, int row );
void     util_wait_microseconds      (int num_microseconds);
void     util_wait_seconds           (int num_seconds);


// PCIe CSR register access functions
int      pci_reg_write_offset        (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar,
uint32_t addr_offset, uint32_t value );
uint32_t pci_reg_read_offset         (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar,
uint32_t addr_offset );
int      pci_reg_set_bits_offset     (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar,
uint32_t addr_offset, int start_bit, int stop_bit );
int      pci_reg_clear_bits_offset   (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar,
uint32_t addr_offset, int start_bit, int stop_bit );

// PCI specific functions
void     pci_read_reg_ctrl_version   (ACX_PCIE_dev_handle *device, ACX_BAR_handle *reg_ctrl_bar);
bool     pci_link_is_up              (ACX_PCIE_dev_handle *device);

// DMA specific functions
void      dma_build_data_descriptor  (DMADataDescriptor *desc, uint32_t size, uint64_t sar,
uint64_t dar);
void      dma_build_link_descriptor  (DMALinkDescriptor *desc, uint64_t ptr_phys_addr);
int       dma_init                   (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar,
PartName part, PCIeCoreNum core);
void      dma_config                 (ACX_PCIE_dev_handle *device, DmaCommand_t &p_dma_inst);
void      dma_start                  (ACX_PCIE_dev_handle *device, DmaCommand_t &p_dma_inst);
void      dma_halt                   (ACX_PCIE_dev_handle *device, DmaCommand_t &p_dma_inst);
DmaStatus dma_get_status             (ACX_PCIE_dev_handle *device, DmaCommand_t &p_dma_inst);
DmaStatus dma_wait                   (ACX_PCIE_dev_handle *device, DmaCommand_t &command, int
timeout_in_seconds);

// ATU specific functions
void      atu_get_context            (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar,
PCIeCoreNum pcie_core, ATUContext &context);
void      atu_find_regions           (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar,
PCIeCoreNum pcie_core, int bar_num, std::vector<ATURegion> &regions);
void      atu_get_region             (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar,
PCIeCoreNum pcie_core, int region_num, ATURegion &region);
void      atu_put_region             (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar,
PCIeCoreNum pcie_core, ATURegion &region);
```

util_calc_nap_absolute_addr()

Description

Calculate the full device 42-bit address of a NAP, placed in the 2D NoC, given the row and column coordinates. This function is primarily intended for use when a DMA transfer is required between a NAP and a host using the device address. The source code for this function is available in `/src/Achronix_PCI.cpp`.

Call

```
uint64_t util_calc_nap_absolute_addr (PartName part, int col, int row);
```

Arguments

Table 4: ***Utility Calculate NAP Absolute Address Function Arguments***

Type	Argument	Description
`Partname`	`part`	Device partname. Supported values are `AC7t1500ES0`.
`int`	`col`	Column address. Column values start from 1 (not 0).
`int`	`row`	Row address. Row values start from 1 (not 0).

Return Value

Returns the absolute 2D NoC 42-bit address of the NAP.

util_wait_microseconds()

Description

Non-blocking function to sleep for a defined number of microseconds (μS). The source code for this function is available in `/src/Achronix_PCI.cpp`.

Call

```
void util_wait_microseconds (int num_microseconds);
```

Arguments

Table 5: *Utility Wait Microseconds Function Arguments*

Type	Argument	Description
`int`	`num_microseconds`	Number of microseconds to sleep.

Return Value

The function does not have a return value.

util_wait_seconds()

Description

Non-blocking function to sleep for a defined number of seconds. The source code for this function is available in `/src/Achronix_PCI.cpp`.

Call

```
void util_wait_seconds (int num_seconds);
```

Arguments

Table 6: ***Utility Wait Seconds Function Arguments***

Type	Argument	Description
`int`	`num_seconds`	Number of seconds to sleep.

Return Value

The function does not have a return value.

pci_reg_write_offset()

Description

Write to a register in the device, using a 32-bit offset to the required PCIe BAR region.

Call

```
int pci_reg_write_offset (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, uint32_t
addr_offset, uint32_t value);
```

Arguments

Table 7: *PCI Register Write Offset Function Arguments*

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	PCIe BAR that references the register location. This must be the BAR set to the configuration status registers in the PCIe DBI space.
`uint32_t`	`addr_offset`(1)	32-bit offset to the BAR base address.
`uint32_t`	`value`	Value to be written to the register.

Table Notes

1. Currently there is a restriction with the PCIe BAR size to 64MB. Therefore, `addr_offset` is limited to a maximum value of `0x03ff_ffff`.

Return Value

Returns a positive value indicating the number of 32-bit writes (1) if it completed successfully. Returns a negative value if unsuccessful.

pci_reg_read_offset()

Description

Write to a register in the device, using a 32-bit offset to the required PCIe BAR region.

Call

```
uint32_t pci_reg_read_offset (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, uint32_t
addr_offset);
```

Arguments

Table 8: ***PCI Register Read Offset Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	PCIe BAR that references the register location. This must be the BAR set to the configuration status registers in the PCIe DBI space.
`uint32_t`	`addr_offset`	32-bit offset to the BAR base address.

Table Notes

1. Currently there is a restriction with the PCIe BAR size to 64MB. Therefore, `addr_offset` is limited to a maximum value of `0x03ff_ffff`.

Return Value

Returns the 32-bit value of the register.

pci_reg_set_bits_offset()

Description

Set a range of bits in a register in the device to `1'b1`, using a 32-bit offset to the required PCIe BAR region. The function performs a read-modify-write sequence on the register.

Call

```
int pci_reg_set_bits_offset (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, uint32_t
addr_offset, int start_bit, int stop_bit);
```

Arguments

Table 9: ***PCI Register Set Bits Offset Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	PCIe BAR that references the register location. This must be the BAR set to the configuration status registers in the PCIe DBI space.
`uint32_t`	`addr_offset`(1)	32-bit offset to the BAR base address.
`int`	`start_bit`	Highest bit to be set. Must be in the range 0–31. Must be >= `stop_bit`.
`int`	`stop_bit`	Lowest bit to be set. Must be in the range 0–31. Must be <= `stop_bit`.

Table Notes

1. Currently there is a restriction with the PCIe BAR size to 64MB. Therefore, `addr_offset` is limited to a maximum value of `0x03ff_ffff`.

Return Value

Returns a positive value indicating the number of 32-bit writes (1) if it completed successfully. Returns a negative value if unsuccessful.

pci_reg_clear_bits_offset()

Description

Clear a range of bits in a register in the device to `1'b0`, using a 32-bit offset to the required PCIe BAR region. The function performs a read-modify-write sequence on the register.

Call

```
int pci_reg_clear_bits_offset (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, uint32_t
addr_offset, int start_bit, int stop_bit);
```

Arguments

Table 10: ***PCI Register Clear Bits Offset Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	PCIe BAR that references the register location. This must be the BAR set to the configuration status registers in the PCIe DBI space.
`uint32_t`	`addr_offset`(1)	32-bit offset to the BAR base address.
`int`	`start_bit`	Highest bit to be cleared. Must be in the range 0–31. Must be >= `stop_bit`.
`int`	`stop_bit`	Lowest bit to be cleared. Must be in the range 0–31. Must be <= `stop_bit`.

Table Notes

1. Currently there is a restriction with the PCIe BAR size to 64MB. Therefore, `addr_offset` is limited to a maximum value of `0x03ff_ffff`.

Return Value

Returns a positive value indicating the number of 32-bit writes (1) if it completed successfully. Returns a negative value if unsuccessful.

pci_read_reg_ctrl_version()

Description

Utility function to display the values of the version registers within a register control block.

Call

```
int pci_read_reg_ctrl_version (ACX_PCI_dev_handle *device, ACX_BAR_handle *reg_ctrl_bar);
```

Arguments

Table 11: *PCI Read Register Control Version Function Arguments*

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`reg_ctrl_bar`	PCIe BAR that references the register control block.

Return Value

Returns 0.

pci_link_is_up()

Description

Utility function to indicate if the PCIe device is correctly enumerated and available for access. The function ensures that the vendor ID register can be read via DBI over the CSR BAR.

Call

```
bool pci_link_is_up (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar);
```

Arguments

Table 12: ***PCI Link Is Up Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	PCIe BAR that references the register location. Must be the BAR set to the configuration status registers in the PCIe DBI space.

Return Value

Returns true if the PCIe core responds correctly, or false if an error is detected.

dma_build_data_descriptor()

Description

Populates a DMA data descriptor in a `DMADescriptorList`. Used for linked-list-mode DMA operation. The source code for this function is available in `<achronix_SDK>/src/Achronix_PCI.cpp`.

Call

```
void dma_build_data_descriptor (DMADataDescriptor *desc, uint32_t size, uint64_t sar, uint64_t
dar);
```

Arguments

Table 13: ***DMA Build Data Descriptor Function Arguments***

Type	Argument	Description
`DMADataDescriptor*`(1) (2)	`desc`	Pointer to the descriptor.
`uint32_t`	`size`	Size of the transfer in bytes.
`uint64_t`	`sar`	Source address (can be either host or device).
`uint64_t`	`dar`	Destination address (can be either host or device).

Table Notes

1. The descriptor must already be defined (normally as part of a `DMADescriptorList`).
2. The direction of the DMA transfer is not defined in the descriptor. The direction is set by `dma_config()`. It is therefore important that `sar` and `dar` are set correctly in every descriptor with respect to host and device addresses to be consistent with the subsequent direction set by `dma_config()`.

Return Value

The function does not have a return value.

dma_build_link_descriptor()

Description

Populates the DMA link descriptor that terminates each `DMADescriptorList`. Used for linked-list DMA operation. The `DMALinkDescriptor` that terminates each `DMADescriptorList` is filled in by the `DMADescriptorList` constructor to point back to the first descriptor in list list. This function is only needed when building multiple linked sets of `DMADescriptorLists`. The source code for this function is available in `/src/Achronix_PCI.cpp`.

Call

```
void dma_build_link_descriptor (DMALinkDescriptor *desc, uint64_t ptr_phys_addr);
```

Arguments

Table 14: *DMA Build Link Descriptor Function Arguments*

Type	Argument	Description
`DMALinkDescriptor*`(1) (2)	`desc`	Pointer to the descriptor.
`uint64_t`	`ptr_phys_addr`	Start address of the next block of link-list descriptors. If the start of the current block of descriptors is used, this address acts as an end-of-list for the current linked list, causing the DMA to complete. Defined as the device physical address within the 42-bit 2D NoC memory space.

Table Notes

1. The descriptor must have already been defined, normally as part of a `DMADescriptorList`.
2. The direction of the DMA transfer is not defined in the descriptor. The direction is set by `dma_config()`. It is therefore important that `sar` and `dar` are set correctly in every descriptor with respect to host and device addresses to be consistent with the subsequent direction set by `dma_config()`.

Return Value

The function does not have a return value.

dma_init()

Description

Initializes the PCIe DMA engine. Sets the arbitration weights for each of the four DMA channels to the same value. Must be called at least once before any DMA transactions are initiated.

Warning!

Normally, this function should only be called once during program execution. Calling this during DMA operation could cause the core to enter an unknown state.

Call

```
int dma_init (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, PartName part, PCIeCoreNum
core);
```

Arguments

Table 15: ***DMA Initialize Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`PartName`	`part`	Device partname. Currently the only supported option is `AC7t1500ES0`.
`PCIeCoreNum`	`core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.

Return Value

Returns 0.

dma_config()

Description

Configure the PCIe DMA for a transfer. The `DmaCommand` structure passed to this function contains the source and destination addresses, the size of the transfer and the direction. Alternately, if linked-lists are being used, the structure includes the address of the start of the linked list in the device 42-bit 2D NoC address space.

Note

This function does not start a DMA transfer.

Call

```
void dma_config (ACX_PCI_dev_handle *device, DmaCommand &dma_command);
```

Arguments

Table 16: ***DMA Configure Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`DmaCommand&`	`dma_command`	Reference to a DMA command struct.

Return Value

The function does not have a return value.

dma_start()

Description

Starts a DMA transfer previously configured with a call to `dma_config()`.

Call

```
void dma_start (ACX_PCI_dev_handle *device, DmaCommand &dma_command);
```

Arguments

Table 17: *DMA Start Function Arguments*

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`DmaCommand&`	`dma_command`	Reference to the DMA command struct. This command struct must have been previously processed by `dma_config()`.

Return Value

The function does not have a return value.

dma_halt()

Description

Halts the currently running PCIe DMA transfers defined by the DMA command instance. Use this only if the DMA transaction has timed out and needs to be aborted.

Call

```
void dma_halt (ACX_PCI_dev_handle *device, DmaCommand &dma_command);
```

Arguments

Table 18: ***DMA Halt Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`DmaCommand&`	`dma_command`	Reference to a DMA command.

Return Value

The function does not have a return value.

dma_print_stats()

Description

Prints a histogram of the DBI write retry stats to stdout. The statistics values are cleared whenever `dma_init()` is called.

Call

```
void dma_print_stats (void);
```

Arguments

The function takes no arguments.

Return Value

The function does not have a return value.

dma_get_status()

Description

Get the status of the currently running PCIe DMA defined by the DMA command structure.

Call

```
DmaStatus dma_get_status (ACX_PCI_dev_handle *device, DmaCommand &dma_command);
```

Arguments

Table 19: ***DMA Get Status Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`DmaCommand&`	`dma_command`	Reference to a DMA command struct.

Return Value

Returns one of the values for `DMAstatus` defined in the `DmaStatus` enum.

dma_wait()

Description

Begins polling the status of a currently running DMA transfer, initiatied with `dma_start()`, at a set interval, and returns to the caller when the transfer is complete.

Note

This function is multi-threaded and non-blocking.

Call

```
DmaStatus dma_wait (ACX_PCI_dev_handle *device, DmaCommand &dma_command, int timeout_in_seconds);
```

Arguments

Table 20: ***DMA Wait Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`DmaCommand&`	`dma_command`	Reference to a DMA command struct.
`int`	`timeout_in_seconds`	Maximum time to wait for DMA to complete. If `timeout_in_seconds` is exceeded, the function returns to the calling function with an error return value.

Return Value

Returns one of the predefined values for `DmaStatus` as defined in `Acxronix_PCI.h`. If the status is not `acxsdk::DMA_COMPLETE`, then the transaction did not complete successfully. Call `dma_halt()` to force the currently running transaction to end correctly.

atu_get_context()

Description

Returns the complete context (configuration register values) of the address translation unit. Results are returned in the `ATUContext` class, which is a vector of 100 `ATURegion` classes. This function can be used to view the current ATU configuration, and then modify it with the `atu_put_region()` function.

Call

```
void atu_get_context (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, PCIeCoreNum core,
ATUContext &context);
```

Arguments

Table 21: ***ATU Get Context Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`PCIeCoreNum`	`core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.
`ATUContext&`	`context`	A reference to an `ATUContext` class to be filled in with the current values of all of the ATU configuration registers.

Return Value

The function does not have a return value.

atu_find_regions()

Description

Finds all regions that cover a given BAR and returns their context in a vector of `ATURegion` classes.

Call

```
void atu_find_regions (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, PCIeCoreNum core, int
bar_num, std::vector<ATURegion> &regions);
```

Arguments

Table 22: ***ATU Find Regions Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`PCIeCoreNum`	`core`	Selects appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.
`int`	`bar_num`	Integer index (0–5) of the BAR number to be found.
`std::vector<ATURegion>&`	`regions`	A reference to a vector of all `ATURegions` that cover BAR number `bar_num`.

Return Value

The function does not have a return value.

atu_get_region()

Description

Gets a given ATU region, and returns its context in an `ATURegion` class.

Call

```
void atu_get_region (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, PCIeCoreNum pcie_core,
int region_num, ATURegion &region);
```

Arguments

Table 23: ***ATU Get Region Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.
`int`	`region_num`	The index (0–99) of the region to be returned.
`ATURegion&`	`region`	A reference to an `ATURegion` class to be filled in with the context of the region specified by `region_num`.

Return Value

The function does not have a return value.

atu_put_region()

Description

Modifies the ATU configuration registers of a given region using the context specified in a given `ATURegion` class.

Call

```
void atu_put_region (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, PCIeCoreNum pcie_core,
ATURegion &region);
```

Arguments

Table 24: ***ATU Put Region Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`PCIeCoreNum`	`pcie_core`	The index (0–99) of the region to be configured.
`ATURegion&`	`region`	A reference to an `ATURegion` class. The context of the class is copied into the configuration registers of the region specified by `region_num`.

Return Value

The function does not have a return value.

msix_is_enabled()

Description

Queries the PCIe device whether the MSI-X interripts are enabled or disabled

Call

```
bool msix_is_enabled (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar, PCIeCoreNum
pcie_core);
```

Arguments

Table 25: *MSI-X Is Enabled Function Arguments*

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.

Return Value

Returns true if MSI-X interrupts are enabled, otherewise returns false.

msix_get_table_size()

Description

Queries the PCIe device for the number of entries in the MSI-X vector and PBA tables (the number of supported interrupt vectors).

Call

```
int msix_get_table_size (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar, PCIeCoreNum
pcie_core);
```

Arguments

Table 26: ***MSI-X Get Table Size Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.

Return Value

Returns the integer number of entries in the MSI-X vector and PBA tables.

msix_get_context()

Description

Queries the PCIe device to read the values of all MSI-X configuraton registers, also called the MSI-X context. Values of the configuration registers are returned in the `MSIXContext` class. This function is intended for low-level debugging of hardware resources. It is not needed during ordinary application execution.

Call

```
void msix_get_context (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar, PCIeCoreNum
pcie_core, MSIXContext &context);
```

Arguments

Table 27: *MSI-X Get Context Function Arguments*

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.
`MSIXContext&`	`context`	On return, a structure holding all of the MSI-X configuration register settings.

Return Value

The function does not have a return value. A structure containing all of the MSI-X configuration register values is written to the context argument.

msix_get_vector()

Description

Queries the PCIe device to read the enumeration state associated with the interrupt vector at a given vector index. The state includes the vector address, the message data, and the mask bit.

Call

```
void msix_get_vector (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar, ACX_BAR_handle
*msix_bar, PCIeCoreNum pcie_core, int index, uint64_t *message_address, uint32_t *message_data,
int *mask_bit);
```

Arguments

Table 28: ***MSI-X Get Vector Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`ACX_BAR_handle*`	`msix_bar`	BAR that references the MSI-X vector table and pending bit array.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.
`int`	`index`	Index of the interrupt vector being queried.
`uint64_t*`	`message_address`	On return, contains the 64-bit message address at the given index.
`uint32_t*`	`message_data`	On return, contains the 32-bit message data at the given index.
`int*`	`mask_bit`	On return, contains the 1-bit per-vector mask bit (asserted or de-asserted).

Return Value

The function does not have a return value. See arguments for return data.

msix_get_pending_bit()

Description

Queries the PCIe device to read the pending bit associated with an interrupt vector at a given index.

Call

```
int msix_get_pending_bit (ACX_PCIE_dev_handle *device, ACX_BAR_handle *csr_bar, ACX_BAR_handle
*msix_bar, PCIeCoreNum pcie_core, int index, int *pending_bit);
```

Arguments

Table 29: ***MSI-X Get Pending Bit Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`ACX_BAR_handle*`	`msix_bar`	BAR that references the MSI-X vector table and pending bit array.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.
`int`	`index`	Index of the interrupt vector being queried.
`int*`	`pending_bit`	On return, contains the single pending bit for the vector at the given index.

Return Value

Returns the value of the pending bit for the interrupt at the given index.

msix_set_function_mask()

Description

Sets the global per-function mask bit.

Call

```
void msix_set_function_mask (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, PCIeCoreNum
pcie_core, int value);
```

Arguments

Table 30: ***MSI-X Set Function Mask Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.
`int`	`value`	The 1-bit value to set for the global per-function interrupt mask (0 = de-asserted, 1 = asserted).

Return Value

The function does not have a return value.

msix_set_vector_mask()

Description

Sets the per-vector interrupt mask bit for the vector at the given index.

Call

```
void msix_set_vector_mask (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, ACX_BAR_handle
*msix_bar, PCIeCoreNum pcie_core, int index, int value);
```

Arguments

Table 31: ***MSI-X Set Vector Mask Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`ACX_BAR_handle*`	`msix_bar`	BAR that references the MSI-X vector table and pending bit array.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.
`int`	`index`	The index of the vector being set.
`int`	`value`	The 1-bit value to set for the per-vector interrupt mask (0 = de-asserted, 1 = asserted).

Return Value

The function does not have a return value.

msix_print_vectors()

Description

Prints the state associated with all of the interrupt vectors to the console for debugging purposes. The per-vector state includes the message address, the message data, and the mask bit.

Call

```
void msix_print_vectors (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, ACX_BAR_handle
*msix_bar, PCIeCoreNum pcie_core);
```

Arguments

Table 32: ***MSI-X Print Vectors Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`ACX_BAR_handle*`	`msix_bar`	BAR that references the MSI-X vector table and pending bit array.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are `PCIE_0` and `PCIE_1`.

Return Value

The function does not have a return value.

msix_print_pending_bits()

Description

Prints the interrupt Pending Bit Array (PBA) to the console for debugging purposes.

Call

```
void msix_print_pending_bits (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, ACX_BAR_handle
*msix_bar, PCIeCoreNum pcie_core);
```

Arguments

Table 33: ***MSI-X Print Pending Bits Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`ACX_BAR_handle*`	`msix_bar`	BAR that references the MSI-X vector table and pending bit array.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are PCIE_0 and PCIE_1.

Return Value

The function does not have a return value.

msix_interrupt()

Description

Triggers an interrupt using the vector at the given index. Interrupts are normally triggered from the hardware in response to some asynchronous events, but this function allows interrupts to be triggered from the host software for debugging and testing purposes.

Call

```
void msix_interrupt (ACX_PCI_dev_handle *device, ACX_BAR_handle *csr_bar, PCIeCoreNum pcie_core,
int message_id);
```

Arguments

Table 34: ***MSI-X Interrupt Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`ACX_BAR_handle*`	`csr_bar`	BAR that references the configuration status registers in the PCIe DBI space.
`PCIeCoreNum`	`pcie_core`	Selects the appropriate PCIe core. Options are PCIE_0 and PCIE_1.
`int`	`message_id`	The index of the interrupt vector being triggered.

Return Value

The function does not have a return value.

msix_interrupt_wait()

Description

Waits for an interrupt to be triggered at the given interrupt vector index, and then returns to the caller.

Call

```
MsixStatus msix_interrupt_wait (ACX_PCI_dev_handle *device, int message_id, unsigned int
timeout_ms, unsigned int *interrupt_count);
```

Arguments

Table 35: *MSI-X Interrupt Wait Function Arguments*

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`int`	`message_id`	The index of the interrupt vector being queried.
`unsigned int`	`timeout_ms`	The number of milliseconds to wait for an interrupt to occur. A value of "0" indicates wait forever.
`unsigned int*`	`interrupt_count`	A pointer to an unsigned integer to contain the number of interrupts received by the driver since it was loaded.

Return Value

Returns one of the values for MSI-X status defined in the `MsixStatus` enum.

msix_cancel_wait()

Description

Cancel a call to `msix_interrupt_wait()` with the specifed `message_id` in a different thread. This function does not wait for the waiting thread to be finished with the `msix_interrupt_wait()` function call.

Call

```
void msix_cancel_wait (ACX_PCI_dev_handle *device, int message_id);
```

Arguments

Table 36: ***MSI-X Cancel Wait Function Arguments***

Type	Argument	Description
`ACX_PCI_dev_handle*`	`device`	Pointer to the PCIe device.
`int`	`message_id`	The index of the interrupt vector being queried.

Return Value

The function does not have a return value.

Chapter - 9: SDK Structures

DmaCommand_t

Description

The DmaCommand_t structure is used to specify the parameters of a DMA transaction when calling dma_config() and dma_start(). The structure can be used in two modes: normal and linked-list. In normal mode, all of the parameters are directly specified in the DmaCommand, hence the device_address, host_address, and size_in_bytes fields are populated and the descriptor_list_address is set to NULL. In linked-list mode those three parameters are read from the descriptors, so device_address, host_address, and size_in_bytes are set to NULL and the descrptor_list_address element is populated. See the DMA example source code for more information.

Definition

```
struct DmaCommand_t {
    ACX_BAR_handle*   csr_bar;
    PCIeCoreNum       pcie_core;
    DmaDir            dma_direction;
    int               dma_channel;
    uint64_t          device_address;
    uint64_t          host_address;
    uint64_t          size_in_bytes;
    uint64_t          descriptor_list_address;
    bool              verbosity;
};
```

Fields

Table 37: *DmaCommand_t Structure Fields*

Type	Parameter	Description
ACX_BAR_handle*	csr_bar	A handle to a BAR register mapped to the base of the CSR space in the 42-bit 2D NoC address space.
PCIeCoreNum	pcie_core	An enum that specifies which PCIe core is being programmed. Normally, this is PCIE_1 which is the core connected to the host PC.
DmaDir	dma_direction	An enum describing the transfer direction. Either HOST_TO_DEVICE (a read) or DEVICE_TO_HOST (a write).
int	dma_channel	Specifies which of the DMA channels to program. The Speedster7t FPGA has four independent full-duplex channels (0–3).

Type	Parameter	Description
`uint64_t`	`device_address`	The 42-bit base address for the transfer on the device side (normal mode only).
`uint64_t`	`host_address`	The 64-bit address for the transfer on the host side. This must be the physical (not virtual) address of a DMA buffer (normal mode only).
`uint64_t`	`size_in_bytes`	The size of the transfer specified as the number of bytes (normal mode only).
`uint64_t`	`descriptor_list_address`	The 42-bit base address of a DMA data descriptor (linked-list mode only).
`bool`	`verbosity`	Setting this flag to a non-zero value increases the detail of the output debug information.

DMADataDescriptor

Description

This struct consists of six 32-bit parameters specifying the meta-parameters for a DMA transaction. The parameters can be allocated in blocks of contiguous descriptors using the `DMADescriptorList` class described in the following example.

Definition

```
struct DMADataDescriptor {
    uint32_t control;
    uint32_t size;
    uint32_t sar_low;
    uint32_t sar_high;
    uint32_t dar_low;
    uint32_t dar_high;
};
```

Fields

Table 38: *DMADataDescriptor Structure Fields*

Type	Parameter	Description
`uint32_t`	`control`	A 32-bit control register. Only bits [4:0] are currently in use. See the file `Achronix_PCI.cpp` for usage.
`uint32_t`	`size`	The size of the transaction specified as the number of bytes.
`uint32_t`	`sar_low`	The lower 32 bits of the DMA source address.
`uint32_t`	`sar_high`	The upper 32 bits of the DMA source address.
`uint32_t`	`dar_low`	The lower 32 bits of the DMA destination address.
`uint32_t`	`dar_high`	The upper 32 bits of the DMA destination address.

DMALinkDescriptor

Description

This struct consists of six 32-bit parameters used in DMA linked-list mode. When allocating a block of descriptors using a `DMADescriptorList` class (described in the following example), the last descriptor in a contiguous block must be a link descriptor. The `ptr` field in a link descriptor points to the first descriptor in a neighboring descriptor list. The last link descriptor list in a linked-list should point back to the beginning of the first data descriptor in the list. The `DMALinkDescriptor` struct is the same size as the `DMADataDescriptor`, however three of the 32-bit fields are unused.

Definition

```
struct DMALinkDescriptor {
    uint32_t control;
    uint32_t unused_0;
    uint32_t ptr_low;
    uint32_t ptr_high;
    uint32_t unused_1;
    uint32_t unused_2;
};
```

Fields

Table 39: *DMALinkDescriptor Structure Fields*

Type	Parameter	Description
`uint32_t`	`control`	A 32-bit control register. Only bits [2:0] are currently in use. See the file `Achronix_PCI.cpp` for usage.
`uint32_t`	`unused_0`	This field is not currently in use.
`uint32_t`	`ptr_low`	The lower 32 bits of the next `DMADataDescriptor` in a linked-list chain.
`uint32_t`	`ptr_high`	The upper 32 bits of the next `DMADataDescriptor` in a linked-list chain.
`uint32_t`	`unused_1`	This field is not currently in use.
`uint32_t`	`unused_2`	This field is not currently in use.

Chapter - 10: SDK Classes

The SDK library includes the following C++ classes. Classes are defined in the `<achronix_SDK>/include/Achronix_PCI.h` file.

PCIDevice

Description

This is a convenience class consisting of a C++ wrapper around the low-level `acx_pcie_device_open()` function. When called, the class constructor attempts to open the PCIe device with the specified `device_id`. When the class destructor is called, the device is closed before the class is deallocated. The function `get_pci_status()` is used to query whether the device was opened successfully.

Definition

```
class PCIDevice {
public:
    enum DeviceStatus {
        STATUS_OK,
        STATUS_ERROR
    };
public:
    PCIDevice(int device_id);
    ~PCIDevice();
    DeviceStatus get_pci_status() { return _pci_status; }
    int get_device_id() { return _device_id; }
    ACX_PCI_dev_handle *get_device() { return _device; }
    ACX_BAR_handle *get_bar_handle(uint32_t bar_id);
    void print();
    // PCI Reads
    int read_uint8(ACX_BAR_handle *bar, uint64_t offset, uint8_t *buffer, int count);
    int read_uint16(ACX_BAR_handle *bar, uint64_t offset, uint16_t *buffer, int count);
    int read_uint32(ACX_BAR_handle *bar, uint64_t offset, uint32_t *buffer, int count);
    int read_uint64(ACX_BAR_handle *bar, uint64_t offset, uint64_t *buffer, int count);
    // PCI Writes
    int write_uint8(ACX_BAR_handle *bar, uint64_t offset, uint8_t *buffer, int count);
    int write_uint16(ACX_BAR_handle *bar, uint64_t offset, uint16_t *buffer, int count);
    int write_uint32(ACX_BAR_handle *bar, uint64_t offset, uint32_t *buffer, int count);
    int write_uint64(ACX_BAR_handle *bar, uint64_t offset, uint64_t *buffer, int count);
};
```

Member Functions

Table 40: *PCIDevice Class Member Functions*

Return Type	Function	Description
`void`	`PCIDevice`	Constructor. Opens the PCI device specified by the `device_id` and retains a handle to the device obtainable with the `get_device()` function.
`void`	`~PCIDevice`	Descructor. Closes the PCI device if open.
`DeviceStatus`	`get_pci_status`	Returns an enum value indicating if the device was opened successfully.
`int`	`get_device_id`	Returns the `device_id` passed into the constructor.
`ACX_PCI_dev_handle*`	`get_device`	Returns a handle to the underlying PCIe device object if opened successfully.
`ACX_BAR_handle*`	`get_bar_handle`	Returns a handle to the underlying PCIe base address register if the device was opened successfully.
`void`	`print`	Displays metadata about the device on the console, including the device and vendor ID strings and the BAR configurations.
`int`	`read_uint8`	Read an 8-bit unsigned integer from the device through a BAR.
`int`	`read_uint16`	Read a 16-bit unsigned integer from the device through a BAR.
`int`	`read_uint32`	Read a 32-bit unsigned integer from the device through a BAR.
`int`	`read_uint64`	Read a 64-bit unsigned integer from the device through a BAR.
`int`	`write_uint8`	Write an 8-bit unsigned integer to the device through a BAR.
`int`	`write_uint16`	Write a 16-bit unsigned integer to the device through a BAR.
`int`	`write_uint32`	Write a 32-bit unsigned integer to the device through a BAR.
`int`	`write_uint64`	Write a 64-bit unsigned integer to the device through a BAR.

DMAHostBuffer

Description

A convenience class consisting of a C++ wrapper around the low-level `acx_dma_malloc()` function. When called, the class constructor allocates a DMA buffer of the given size. When the class destructor is called, the buffer is deallocated. The function, `get_status()`, is used to query whether the buffer was allocated successfully. The functions, `get_phys_addr()` and `get_virt_addr()`, are used to get the physical and virtual addresses (respectively) of the buffer. Functions are provided to clear and fill the buffer with various data patterns.

Definition

```
class DMAHostBuffer {
public:
    enum BufferStatus {
        STATUS_OK,
        STATUS_ERROR
    };
public:
    DMAHostBuffer(PCIDevice *device, uint64_t size_in_bytes);
    ~DMAHostBuffer();
    BufferStatus get_status() { return _status; }
    uint64_t get_size_in_bytes() { return _size_in_bytes; }
    uint64_t get_phys_addr() { return (uint64_t)_phys_addr; }
    uint64_t get_virt_addr() { return (uint64_t)_virt_addr; }
    void clear();
    void fill_random();
    void fill_deadbeef();
    bool compare(DMAHostBuffer&, int verbosity);
};
```

Member Functions

Table 41: *DMAHostBuffer Class Member Functions*

Return Type	Function	Description
`void`	`DMAHostBuffer`	Constructor. Allocates a DMA buffer of the given size.
`void`	`~DMAHostBuffer`	Destructor. Deallocates the DMA buffer.
`BufferStatus`	`get_status`	Returns an enum value indicating if the buffer was allocated successfully.
`uint64_t`	`get_size_in_bytes`	Returns the buffer size. Currently, buffers larger than 4MB are not supported.

Return Type	Function	Description
`uint64_t`	`get_phys_addr`	Returns the physical address of the buffer in the host 64-bit memory space. Use this value when passing the buffer address to the `dma_init()` function through the `DmaCommand` struct `host_address` field.
`uint64_t`	`get_virt_addr`	Returns the virtual address of the buffer in the 64-bit address space of the calling process. Use this address to access the buffer from within the application source code.
`void`	`clear`	Clears the contents of the buffer to all zero values.
`void`	`fill_random`	Fills the buffer with random values for testing.
`void`	`fill_deadbeef`	Fills the buffer with a predictable pattern for testing (`0xDEADBEEF + i`).
`bool`	`compare`	Compares this buffer to another buffer of the same size. Verbosity = 1 displays differences on the screen. Verbosity = 2 displays both buffers side-by-side.

DMADescriptorList

Description

This class defines descriptor lists for DMA transactions when using linked-list mode. A descriptor list is a group of descriptors that are allocated in the same block of adjacent memory locations. Each list consists of (`N+1`) descriptors, as shown in the following figure. The first `N` descriptors are `DMADataDescriptors` (defined above), and the last descriptor is a `DMALinkDescriptor` (also defined above). The link descriptor points to the base address of another `DMADescriptor` list, or back to the start of the first `DMADescriptorList` in the chain if it is the last list in the chain.

When operating in linked-list mode, each data descriptor contains the parameter settings for a single DMA transaction from one block of host memory to device memory, or vice versa. The DMA engine steps through the entire list, using the parameters in each descriptor to initiate DMA transactions one after another, following link descriptors as necessary, until the entire list is consumed. Using this method, a large number of sequential transactions can be performed with only a single call to `dma_init()` and `dma_start()`. When the `DMADescriptorList` constructor is called, a block of N data descriptors and one link descriptor is allocated in host memory and initialized to all-zeros. Each descriptor can be accessed in turn using the square-bracket index operator "`[]`", and a call to `dma_build_data_descriptor()` (described above) populates the descriptor with data. The link descriptor is initialized to refer back to the first data descriptor in the same list, however the `dma_build_link_descriptor()` function can be used to build a chain of more than one list element.

The DMA engine consumes descriptors from device memory, not from the host. Therefore, the descriptors must be transferred from the host to the device before the DMA operation can begin. The descriptors can be stored anywhere in device memory, (GDDR6, DDR4, or a BRAM). The descriptors can be written by either DMA or a sequence of BAR writes. The `DMADescriptorList` provides `get_phys_addr()` and `get_virt_addr()` functions, similar to the `DMAHostBuffer` class, to make DMA transfers of the descriptors straightforward.

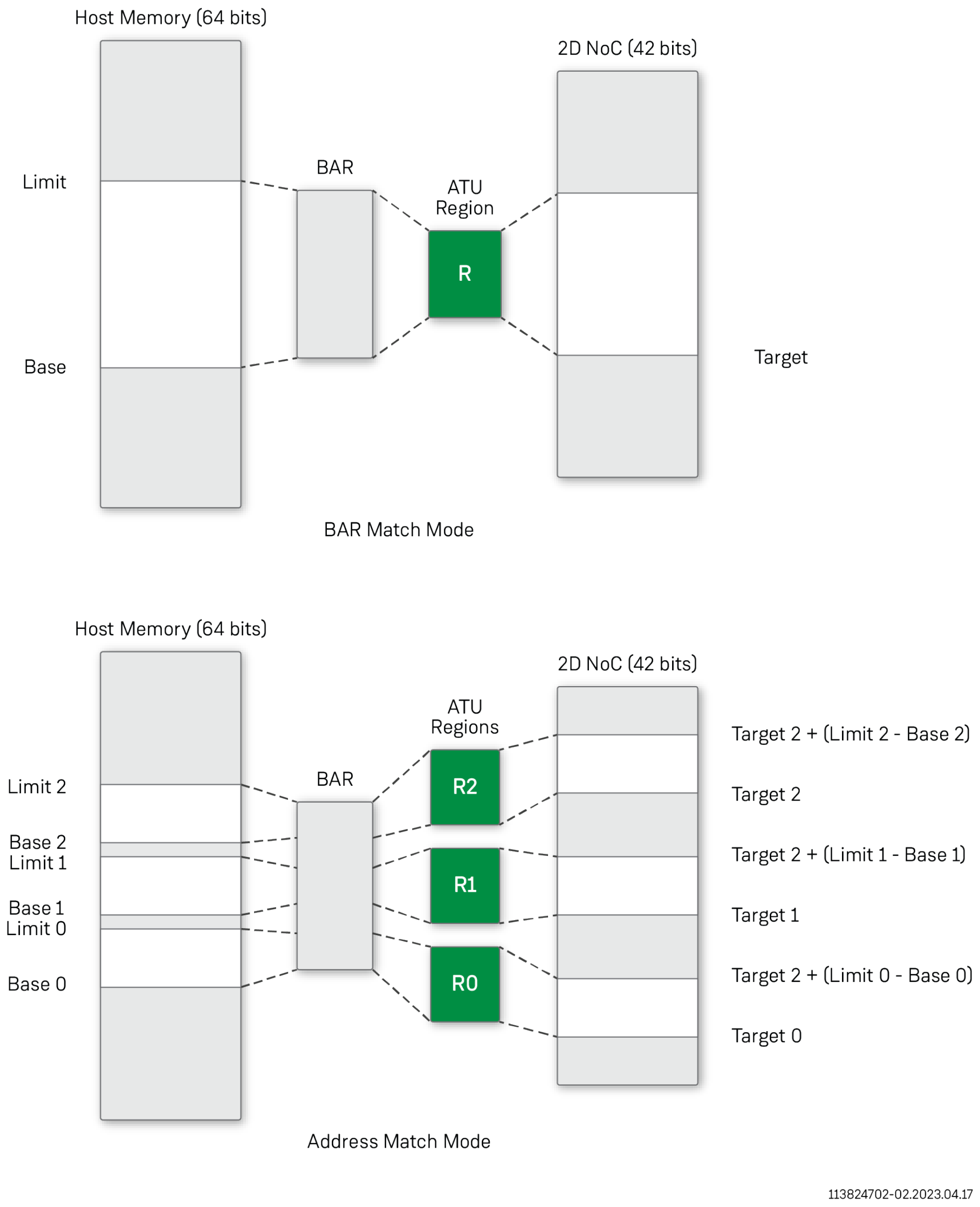


Figure 4: ***DMADescriptorList Structure in Memory***

Definition

```
class DMADescriptorList {
public:
    enum DescriptorStatus {
        STATUS_OK,
        STATUS_ERROR
    };
public:
    DMADescriptorList(PCIDevice *device, int num_descriptors, uint64_t device_phys_addr);
    ~DMADescriptorList();
    DescriptorStatus get_status() { return _status; }
    uint64_t get_size_in_bytes() { return _size_in_bytes; }
    uint64_t get_phys_addr() { return (uint64_t)_phys_addr; }
    uint64_t get_virt_addr() { return (uint64_t)_virt_addr; }
    uint64_t get_device_phys_addr() { return _device_phys_addr; }
    DMADataDescriptor *operator[](int);
    void print(const char *header);
};
```

Member Functions

Table 42: ***DMADescriptorList Class Member Functions***

Return Type	Function	Description
`void`	`DMADescriptorList`	Constructor. Allocates a block of `num_descriptors`, `DMADataDescriptors` and one `DMALinkDescriptor`. The data descriptor is initialized to all zero values, and the link descriptor is initialized to point back to the base address of the list — this is a stopping criteria for the DMA engine. The `device_phys_addr` argument specifies the target address of the descriptor list in host memory.
`void`	`~DMADescriptorList`	Destructor. Deallocates the data and link descriptors.
`DescriptorStatus`	`get_status`	Returns an enum value indicating the success or failure of the descriptor list allocation.
`uint64_t`	`get_size_in_bytes`	Returns the size of the descriptor list in bytes. Use this value when passing the list size to the `dma_init()` function through the `DmaCommand` struct.
`uint64_t`	`get_phys_addr`	Returns the physical address of the descriptor list in the host 64-bit memory space. Use this value when passing the list address to the `dma_init()` function through the `DmaCommand` struct host_address field.
`uint64_t`	`get_virt_addr`	Returns the virtual address of the descriptor list in the 64-bit address space of the calling process. Use this address to access the descriptors in the list from within the C++ application source code.

Return Type	Function	Description
`uint64_t`	`get_device_phys_addr`	Returns the `device_phys_addr` argument passed into the constructor. It specifies the 42-bit 2D NoC address of the list when transferred from host memory into device memory. Use this value when passing the list address to the `dma_init()` function through the `DmaCommand` struct device_address field.
`DMADataDescriptor`	`operator[]`	Allows access to each (`num_descriptors + 1`) of the list descriptors (`num_descriptors`, data descriptors, and one link descriptor) using array semantics. Use this operator when calling the `dma_build_data_descriptor()` and `dma_build_link_descriptor()` functions.
`void`	`print`	Displays the contents of the descriptor list on the console for debugging purposes.

ATUContext

Description

Contains the context (configuration register settings) of all of the ATU registers. The ATU consists of 100 different regions, each with its own set of six configuration registers. The context consists of a vector of 100 `ATURegion` classes (described in the following example).

Definition

```
class ATUContext {
public:
    void print();
public:
    ATURegion _region[100];
};
```

Member Functions

Table 43: *ATUContext Class Member Functions*

Return Type	Function	Description
void	print	Displays the context of each enabled region on the console for visualization and debugging. Regions which are not enabled are skipped.

ATURegion

Description

Contains the context (configuration register settings) of one ATU region. The context consists of nine individual 32-bit registers, each of which can be read with a set of get functions, or written with a set of set functions. The control ("_ctrl_") registers consist of a large number of individual bitfields, each of which has its own (boolean or integer) get and set functions.

Definition

```
class ATURegion {
public:
    ATURegion();
    ATURegion& operator=(const ATURegion& other);
    void print();
    // bitfield getters
    int get_region_num();
    int get_function_num();
    bool get_enabled();
    ATU_MODE get_mode();
    bool get_invert_mode();
    bool get_cfg_shift_mode();
    bool get_fuzzy_type_match_code();
    bool get_vfbar_match_mode_en();
    int get_response_code();
    bool get_single_addr_loc_trans_en();
    bool get_ph_match_en();
    bool get_msg_code_match_en();
    bool get_vf_match_en();
    bool get_func_num_match_en();
    bool get_at_match_en();
    bool get_th_match_en();
    bool get_addr_match_en();
    bool get_td_match_en();
    bool get_tc_match_en();
    bool get_msg_type_match_mode();
    int get_bar_num();
    uint64_t get_base_addr();
    uint64_t get_limit_addr();
    uint64_t get_target_addr();
    // bitfield setters
    void set_region_num(int num);
    void set_function_num(int num);
    void set_enabled(bool val);
    void set_mode(ATU_MODE mode);
    void set_invert_mode(bool val);
    void set_cfg_shift_mode(bool val);
    void set_fuzzy_type_match_code(bool val);
    void set_vfbar_match_mode_en(bool val);
    void set_response_code(int val);
    void set_single_addr_loc_trans_en(bool val);
    void set_ph_match_en(bool val);
    void set_msg_code_match_en(bool val);
    void set_vf_match_en(bool val);
    void set_func_num_match_en(bool val);
    void set_at_match_en(bool val);
    void set_th_match_en(bool val);
    void set_addr_match_en(bool val);
    void set_td_match_en(bool val);
    void set_tc_match_en(bool val);
    void set_msg_type_match_mode(bool val);
    void set_bar_num(int bar_num);
    void set_base_addr(uint64_t addr);
    void set_limit_addr(uint64_t limit);
    void set_target_addr(uint64_t addr);
public:
    int region_num;
    uint32_t iatu_region_ctrl_1_inbound;
    uint32_t iatu_region_ctrl_2_inbound;
    uint32_t iatu_region_ctrl_3_inbound;
    uint32_t iatu_lwr_base_addr_inbound;
    uint32_t iatu_upper_base_addr_inbound;
    uint32_t iatu_lwr_limit_addr_inbound;
    uint32_t iatu_upper_limit_addr_inbound;
    uint32_t iatu_lwr_target_addr_inbound;
    uint32_t iatu_upper_target_addr_inbound;
};
```

Member Functions

Table 44: *ATURegion Class Member Functions*

Return Type	Function	Description
`void`	`print`	Displays the region context on the console for visualization and debugging.
`int`	`get_region_num`	Gets the integer region number of this region, which is also the index of the `ATURegion` in the `ATUContext` class.
`int`	`get_function_num`	Gets the PCIe physical function number of the region.
`bool`	`get_enabled`	Returns true if this region is enabled, otherwise false.
`ATU_MODE`	`get_mode`	Returns the region mode. Either `ATU_BAR_MATCH` or `ATU_ADDRESS_MATCH`.
`bool`	`get_invert_mode`	Reserved for future use.
`bool`	`get_cfgshift_mode`	Reserved for future use.
`bool`	`get_fuzzy_type_match_mode`	Reserved for future use.
`bool`	`get_vfbar_match_mode_en`	Reserved for future use.
`int`	`get_response_code`	Reserved for future use.

Return Type	Function	Description
bool	get_single_addr_loc_trans_en	Reserved for future use.
bool	get_ph_match_en	Reserved for future use.
bool	get_msg_code_match_en	Reserved for future use.
bool	get_vf_match_en	Reserved for future use.
bool	get_fun_num_match_en	Reserved for future use.
bool	get_at_match_en	Reserved for future use.
bool	get_th_match_en	Reserved for future use.
bool	get_addr_match_en	Reserved for future use.
bool	get_td_match_en	Reserved for future use.
bool	get_tc_match_en	Reserved for future use.
bool	get_msg_type_match_mode	Reserved for future use.
int	get_bar_num	Returns the integer index of the BAR. Only valid if the region is in BAR match mode.
uint64_t	get_base_addr	Returns the base (lower) address of the region in the host 64-bit address space. This address must be in a region of host memory assigned to a BAR. Only valid in address match mode.
uint64_t	get_limit_addr	Returns the limit (top) address of the region in the host 64-bit address space. This address must be in a region of host memory assigned to a BAR. It must be a multiple of the minimum region size (64K), so bits [15:4] must be 0xFFF. Only valid in address match mode.
uint64_t	get_target_addr	Returns the base (lower) address of the region in the device 42-bit 2D NoC address space. The device region size must be the same as on the host, so the device limit address is calculated automatically.
void	set_region_num	Sets the region number being specified with this ATURegion class.
void	set_function_num	Sets the physical function number in the region context.
void	set_enabled	Sets the enable bit to true or false in the region context.
void	set_mode	Sets the mode in the region context. Either ATU_BAR_MATCH or ATU_ADDRESS_MATCH.

Return Type	Function	Description
`void`	`set_invert_mode`	Reserved for future use.
`void`	`set_cfg_shift_mode`	Reserved for future use.
`void`	`set_fuzzy_type_match_code`	Reserved for future use.
`void`	`set_vfbar_match_mode_en`	Reserved for future use.
`void`	`set_response_code`	Reserved for future use.
`void`	`set_single_addr_loc_trans_en`	Reserved for future use.
`void`	`set_ph_match_en`	Reserved for future use.
`void`	`set_msg_code_match_en`	Reserved for future use.
`void`	`set_vf_match_en`	Reserved for future use.
`void`	`set_fun_num_match_en`	Reserved for future use.
`void`	`set_at_match_en`	Reserved for future use.
`void`	`set_th_match_en`	Reserved for future use.
`void`	`set_addr_match_en`	Reserved for future use.
`void`	`set_td_match_en`	Reserved for future use.
`void`	`set_tc_match_en`	Reserved for future use.
`void`	`set_msg_type_match_mode`	Reserved for future use.
`void`	`set_bar_num`	Sets the BAR number in the region context.
`void`	`set_base_addr`	Sets the base (lower) address of the region in the host 64-bit address space. This address must be in a region of host memory assigned to a BAR. Only valid in address match mode.
`void`	`set_limit_addr`	Sets the limit (top) address of the region in the host 64-bit address space. This address must be in a region of host memory assigned to a BAR. It must be a multiple of the minimum region size (64K), so bits [15:4] must be `0xFFF`. Only valid in address match mode.
`void`	`set_target_addr`	Sets the base (lower) address of the region in the device 42-bit 2D NoC address space. The device region size must be the same as on the host, so the device limit address is calculated automatically.

MSIXContext

Description

This class holds all of the configuration register settings for the MSI-X controller. The class members can be filled in with the current values of the corresponding configuration register settings by calling the `msix_get_context()` function. Low-level knowledge of the configuration registers, and the individual bit fields of each registers, is required to make use of those values. Not all bit fields are meaningful when read.

Definition

```
class MSIXContext {
   public:
       MSIXContext();
       void print();
   public:
       uint32_t msix_cap_id_next_ctrl_reg;
       uint32_t msix_table_offset_reg;
       uint32_t msix_pba_offset_reg;
       uint32_t msix_address_match_low;
       uint32_t msix_address_match_high;
       uint32_t msix_doorbell;
       uint32_t msix_ram_ctrl;
   };
```

Member Functions

Table 45: *MSIXContext Class Member Functions*

Method	Return Type	Description
`MSIXContext`	`none`	Constructor
`print`	`void`	Print the values for all of the fields.

Members

Table 46: *MSIXContext Class Member Parameters*

Type	Parameter	Description
`uint32_t`	`msix_cap_id_next_ctrl_reg`	The MSI-X capability register.
`uint32_t`	`msix_table_offset_reg`	The offset of the MSI-X vector table from the MSI-X BAR.
`uint32_t`	`msix_pba_offset_reg`	The offset of the MSI-X pending bit array (PBA) from the MSI-X BAR.
`uint32_t`	`msix_address_match_low`	The lower-32 bits of the address match register.
`uint32_t`	`msix_address_match_high`	The upper-32 bits of the address match register.

Type	Parameter	Description
`uint32_t`	`msix_doorbell`	The MSI-X doorbell register.
`uint32_t`	`msix_ram_ctrl`	The MSI-X RAM control register.

Chapter - 11: Driver Translation Resource Handles

ACX_PCIE_dev_handle

Description

Abstract handle to refer to a PCIe device. Used with other translation functions to perform operations on the device referred to by the handle.

ACX_BAR_handle

Description

Abstract handle to refer to a device BAR. Used with other translation functions to perform read/write operations on a specific device BAR which is referred to by the handle.

Note

A BAR handle is associated with the context of the device with which it was created. If working with multiple devices, separate BAR handles must be created for each device.

ACX_DMA_buffer_handle

Description

Abstract handle to refer to a DMA buffer. This handle is only used to hold context for a DMA buffer and is only used in the translation functions for cleanup of a DMA buffer.

Note

A DMA buffer handle is associated with the context of the device with which it was created. DMA handles *cannot* be used across multiple devices. If working with multiple devices, separate DMA buffers must be allocated for each device. If replicating data across multiple devices, the data must be copied into separate DMA buffers and then uploaded to the device associated with that buffer.

Chapter - 12: Driver Translation Functions

acx_pcie_device_open()

Description

opens an Achronix PCIe device for reading and writing.

Call

```
ACX_PCIE_dev_handle* acx_pcie_device_open (uint32_t device_id);
```

Arguments

Table 47: ***Achronix PCIE Device Open Function Arguments***

Type	Argument	Description
`uint32_t`	`device_id`	ID of the device to be opened.

Return Value

Returns a pointer to a handle for a PCIe device.

Note

A device must be opened by calling `acx_pcie_device_open()` before any other translation functions are called. When a device is opened, it must be closed by calling `acx_pcie_device_close()` for proper resource cleanup or before trying to call `acx_pcie_device_open()` again.

acx_pcie_device_close()

Description

Closes an open Achronix PCIe device. The device pointer should be discarded after calling this function.

Call

```
void acx_pcie_device_close (ACX_PCIE_dev_handle *device);
```

Arguments

Table 48: ***Achronix PCIE Device Close Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.

Return Value

The function does not have a return value.

acx_bar_init()

Description

Prepares a device BAR for reading or writing.

Note

When a BAR resource is opened by calling `acx_bar_init()`, it must be closed by calling `acx_bar_cleanup()` before trying to call `acx_bar_init()` again.

Call

```
ACX_BAR_handle* acx_bar_init (ACX_PCIE_dev_handle *device, uint32_t bar_id);
```

Arguments

Table 49: *Achronix BAR Initialize Function Arguments*

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`uint32_t`	`bar_id`	ID of the BAR to be initialized.

Return Value

Returns a pointer to an `ACX_BAR_handle` on success or NULL on failure.

acx_bar_cleanup()

Description

Perofrms any needed cleanup on an `ACX_BAR_handle` pointer. The BAR pointer should be discarded after calling this function.

Call

```
void acx_bar_cleanup (ACX_PCIE_dev_handle *device, ACX_BAR_handle *bar);
```

Arguments

Table 50: ***Achronix BAR Cleanup Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`ACX_BAR_handle*`	`bar`	Pointer to an initilized BAR handle.

Return Value

The function does not have a return value.

acx_get_bar_size()

Description

Gets the amount of mapped memory for a BAR in bytes. The bar size is written into the `size_ptr` argument.

Call

```
int acx_get_bar_size (ACX_PCIE_dev_handle *device, uint32_t bar_id, uint64_t *size_ptr);
```

Arguments

Table 51: *Achronix Get BAR Size Function Arguments*

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`uint32_t`	`bar_id`	ID of the BAR from which to get the size.
`uint64_t*`	`size_ptr`	Pointer to a `uint64_t` into which the size of the bar in bytes is to be written.

Return Value

Returns 0 on success or 1 on failure.

acx_get_bar_start()

Description

Get the host-side address mapping for the start of a BAR.

Note

The address that this function stores in `start_ptr` is not meant to be written to or read from directly by the application. The address returned by this function corresponds to some physical memory managed by the Kernel. Call the reading and writing functions instead.

Call

```
int acx_get_bar_start (ACX_PCIE_dev_handle *device, uint32_t bar_id, uint64_t *start_ptr);
```

Arguments

Table 52: ***Achronix Get BAR Start Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`uint32_t`	`bar_id`	ID of the BAR from which to retrieve the start address.
`uint64_t*`	`start_ptr`	pointer to a `uint64_t` where the start of the BAR is to be written.

Return Value

Returns 0 on success or 1 on failure.

acx_dma_malloc()

Description

Allocates memory for a buffer that can be used in DMA transfers. The application code should use the `virtual_address` pointer to write/read from the buffer. The DMA address is a hardware address that is needed by the DMA engine. For more information on the DMA engine, refer to the DMA Transfers (see page 33) section.

Note

The size of a DMA buffer is limited by the page size of the system (typically 4MB). For larger DMA transfers, refer to the section on DMA Linked List Mode (see page 36).

Call

```
ACX_DMA_buffer_handle* acx_dma_malloc (ACX_PCIE_dev_handle *device, uint64_t size_in_bytes, void
**virtual_address, void **dma_address);
```

Arguments

Table 53: ***Achronix DMA Malloc Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`uint64_t`	`size_in_bytes`	ID of the BAR from which to retrieve the start address.
`void**`	`virtual_address`	A pointer to a void pointer to be loaded with the starting virtual address of the buffer.
`void**`	`dma_address`	A pointer to a void pointer to be loaded with the physical address of the buffer.

Return Value

Returns a pointer to an `ACX_DMA_buffer_handle` on success or NULL on failure.

acx_dma_free()

Description

Frees the memory asscociated with an ACX_DMA_buffer_handle pointer. The ACX_DMA_buffer_handle pointer should be discarded after calling this function.

Call

```
void acx_dma_free (ACX_PCIE_dev_handle *device, ACX_DMA_buffer_handle *dma_mem_handle);
```

Arguments

Table 54: ***Achronix DMA Free Function Arguments***

Type	Argument	Description
ACX_PCIE_dev_handle*	device	Pointer to an opened device handle.
ACX_DMA_buffer_handle*	dma_mem_handle	Pointer to a valid ACX_DMA_buffer_handle.

Return Value

The function does not have a return value.

acx_read_uint8()

Description

Reads single-byte unsigned integers from a BAR into a supplied buffer.

Call

```
int acx_read_uint8 (ACX_PCIE_dev_handle *device, ACX_BAR_handle *bar, uint64_t offset, uint8_t
*buffer, uint32_t count);
```

Arguments

Table 55: ***Achronix Read uint8 Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`ACX_BAR_handle*`	`bar`	Pointer to a valid BAR handle from which to read.
`uint64_t`	`offset`	Offset from the start of the BAR from which to start reading.
`uint8_t*`	`buffer`	Pointer to the start of a buffer into which the data is to be written.
`uint32_t`	`count`	The number of `uint8` values to read from the BAR.

Return Value

Returns the number of bytes read.

acx_read_uint16()

Description

Reads two-byte unsigned integers from a BAR into a supplied buffer.

Call

```
int acx_read_uint16 (ACX_PCIE_dev_handle *device, ACX_BAR_handle *bar, uint64_t offset, uint16_t
*buffer, uint32_t count);
```

Arguments

Table 56: ***Achronix Read uint16 Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`ACX_BAR_handle*`	`bar`	Pointer to a valid BAR handle from which to read.
`uint64_t`	`offset`	Offset from the start of the BAR from which to start reading.
`uint16_t*`	`buffer`	Pointer to the start of a buffer into which the data is to be written.
`uint32_t`	`count`	The number of `uint16` values to read from the BAR.

Return Value

Returns the number of bytes read.

acx_read_uint32()

Description

Reads four-byte unsigned integers from a BAR into a supplied buffer.

Call

```
int acx_read_uint32 (ACX_PCIE_dev_handle *device, ACX_BAR_handle *bar, uint64_t offset, uint32_t
*buffer, uint32_t count);
```

Arguments

Table 57: *Achronix Read uint32 Function Arguments*

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`ACX_BAR_handle*`	`bar`	Pointer to a valid BAR handle from which to read.
`uint64_t`	`offset`	Offset from the start of the BAR from which to start reading.
`uint32_t*`	`buffer`	Pointer to the start of a buffer into which the data is to be written.
`uint32_t`	`count`	The number of `uint32` values to read from the BAR.

Return Value

Returns the number of bytes read.

acx_read_uint64()

Description

Reads eight-byte unsigned integers from a BAR into a supplied buffer.

Call

```
int acx_read_uint64 (ACX_PCIE_dev_handle *device, ACX_BAR_handle *bar, uint64_t offset, uint64_t
*buffer, uint32_t count);
```

Arguments

Table 58: ***Achronix Read uint64 Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`ACX_BAR_handle*`	`bar`	Pointer to a valid BAR handle from which to read.
`uint64_t`	`offset`	Offset from the start of the BAR from which to start reading.
`uint64_t*`	`buffer`	Pointer to the start of a buffer into which the data is to be written.
`uint32_t`	`count`	The number of `uint64` values to read from the BAR.

Return Value

Returns the number of bytes read.

acx_write_uint8()

Description

Writes single-byte unsigned integers from a buffer into a BAR.

Call

```
int acx_write_uint8 (ACX_PCIE_dev_handle *device, ACX_BAR_handle *bar, uint64_t offset, uint8_t
*buffer, uint32_t count);
```

Arguments

Table 59: *Achronix Write unit8 Function Arguments*

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`ACX_BAR_handle*`	`bar`	Pointer to a valid BAR handle from which to read.
`uint64_t`	`offset`	Offset from the start of the BAR to which to start writing.
`uint8_t*`	`buffer`	Pointer to the start of a buffer from which the data is to be read.
`uint32_t`	`count`	The number of `uint8` values to write to the BAR.

Return Value

Returns the number of bytes written.

acx_write_uint16()

Description

Writes two-byte unsigned integers from a buffer into a BAR.

Call

```
int acx_write_uint16 (ACX_PCIE_dev_handle *device, ACX_BAR_handle *bar, uint64_t offset, uint16_t
*buffer, uint32_t count);
```

Arguments

Table 60: *Achronix Write uint16 Function Arguments*

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`ACX_BAR_handle*`	`bar`	Pointer to a valid BAR handle from which to read.
`uint64_t`	`offset`	Offset from the start of the BAR into which to start writing.
`uint16_t*`	`buffer`	Pointer to the start of a buffer from which the data is to be read.
`uint32_t`	`count`	The number of `uint16` values to write to the BAR.

Return Value

Returns the number of bytes written.

acx_write_uint32()

Description

Writes four-byte unsigned integers from a buffer into a BAR.

Call

```
int acx_write_uint32 (ACX_PCIE_dev_handle *device, ACX_BAR_handle *bar, uint64_t offset, uint32_t
*buffer, uint32_t count);
```

Arguments

Table 61: ***Achronix Write uint32 Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`ACX_BAR_handle*`	`bar`	Pointer to a valid BAR handle.
`uint64_t`	`offset`	Offset from the start of the BAR to which to start writing.
`uint32_t*`	`buffer`	Pointer to the start of a buffer from which the data is to be read.
`uint32_t`	`count`	The number of `uint32` values to write to the BAR.

Return Value

Returns the number of bytes written.

acx_write_uint64()

Description

Writes eight-byte unsigned integers from a buffer into a BAR.

Call

```
int acx_write_uint64 (ACX_PCIE_dev_handle *device, ACX_BAR_handle *bar, uint64_t offset, uint64_t
*buffer, uint32_t count);
```

Arguments

Table 62: ***Achronix Write uint64 Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`ACX_BAR_handle*`	`bar`	Pointer to a valid BAR handle.
`uint64_t`	`offset`	Offset from the start of the BAR into which to start writing.
`uint64_t*`	`buffer`	Pointer to the start of a buffer from which the data is to be read.
`uint32_t`	`count`	The number of `uint64` values to write to the BAR.

Return Value

Returns the number of bytes written.

msix_status_to_string()

Description

Converts an `MSixStatus` enum to a C string.

Call

```
const char *msix_status_to_string(MsixStatus);
```

Arguments

Table 63: *MSI-X Status to String Function Arguments*

Type	Argument	Description
`MsixStatus`	`status`	MsixStatus enum value to convert to a string.

Return Value

Returns a pointer to the string representation of the status.

acx_interrupt_wait()

Description

Waits for an interrupt to be triggered at the given interrupt vector index, and then returns to the caller.

Call

```
MsixStatus acx_interrupt_wait (ACX_PCIE_dev_handle *device, uint32_t message_id, unsigned int
timeout_ms, unsigned int* interrupt_count);
```

Arguments

Table 64: ***Achronix Interrupt Wait Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`uint32_t`	`message_id`	The index of the interrupt vector being queried.
`unsigned int`	`timeout_ms`	The number of milliseconds to wait for an interrupt to occur. A value of zero indicates wait forever.
`unsigned int*`	`interrupt_count`	A pointer to an unsigned integer to contain the number of interrupts received by the driver since it was loaded.

Return Value

Returns one of the values for MSI-X status defined in the MsixStatus enum.

acx_cancel_wait()

Description

Cancel a call to `acx_interrupt_wait()` with the specifed `message_id` in a different thread. This function does not wait for the waiting thread to be finished with the `msix_interrupt_wait()` function call.

Call

```
void acx_cancel_wait (ACX_PCIE_dev_handle *device, uint32_t message_id);
```

Arguments

Table 65: ***Achronix Cancel Wait Function Arguments***

Type	Argument	Description
`ACX_PCIE_dev_handle*`	`device`	Pointer to an opened device handle.
`uint32_t`	`message_id`	The index of the interrupt vector that is being cancelled.

Return Value

This function does not have a return value.

Chapter - 13: Porting Guide

This section contains information about backward compatibility, and when backward compatibility is not maintained, what changes might be required to port existing software to new versions of this SDK.

Porting to Version 1.9

The release of the Achronix Software Development Kit version 1.9 brought support for a native Achronix device driver, as well as the original BittWare driver. Supporting both required the introduction of the Achronix driver translation API to wrap around the low-level driver functions. For more information on the driver translation API, refer to the Software Stack (see page 16) and Driver Translation Functions (see page 94) sections. The inclusion of the driver translation API means that some older SDK code is no longer compatible with release version 1.9. The following changes are required.

ACX Resource Handles

In order to support multiple device driver APIs, all driver resources are now tracked and controlled via the use of abstracted resource handles. There are 3 basic resource handle types:

```
Handle Types

ACX_PCIE_dev_handle
ACX_BAR_handle
ACX_DMA_buffer_handle
```

For more information on these handles, please refer to the Driver Translation Resource Handles (see page 93) section.

Older SDK code used the BittWare primitive resource types. In order to upgrade to version 1.9, the following general type conversions need to be applied:

```
Handle conversions

HBwpciDevice -> ACX_PCIE_dev_handle
BwpciMs -> ACX_BAR_handle // When a Bittware memory space reffers to a BAR
BwpciMs -> ACX_DMA_buffer_handle // When a bittware memory space reffers to a DMA buffer
```

Since the BittWare memory space primitive, `BwpciMs`, is able to refer to both a BAR and a buffer, some care is necessary when converting types. Make sure to choose the `ACX_BAR_handle` or `ACX_DMA_buffer_handle`, as appropriate. For reference on which SDK functions return or expect BAR or DMA handles, consult the SDK Functions (see page 39) section.

BARs and the acxsdk::PCIDevice Object

In older versions of the SDK, BARs could be referred to at any point in the application via the globally defined `BwpciMsPreDefined` enum. With the introduction of the driver translation API, this is no longer possible. Ownership and lifetime management of the BAR resources is now the responsibility of the `acxsdk::PCIDevice` object. To get a BAR resource handle, the `acxsdk::PCIDevice::get_bar_handle()` function must be used. This also means that an `acxsdk::PCIDevice` object must be constructed before attempting to get a BAR resource handle. The lifetime of the BAR resource handle is connected to the lifetime of the owning `acxsdk::PCIDevice` object. If the owning `acxsdk::PCIDevice` object is destroyed, all outstanding BAR resource handles that were gathered from that object are no longer valid and should not be used.

The following is a code conversion example:

BAR handle conversions

```
//Pre SDK 1.9 code
BwpciMs reg_ctrl_bar = BW_MS_BAR0;
BwpciMs csr_bar = BW_MS_BAR3;

//Post SDK 1.9 code
acxsdk::PCIDevice device(device_id); // a device must be constructed before getting a BAR
resource handle
ACX_BAR_handle *reg_ctrl_bar = device.get_bar_handle(0);
ACX_BAR_handle *csr_bar = device.get_bar_handle(3);
```

Part Name Removals

The part names for the Speedster7t AC7t1550ES1 and AC7t1500ES1 FPGAs have been removed from the `PartName` enum. When referring to these parts, use the same name without the "ES1" designation as follows:

Handle conversions

```
AC7t1550ES1 -> AC7t1550
AC7t1500ES1 -> AC7t1500
```

Miscellaneous

Several of the software examples that are shipped with the SDK make use of the `acxsdk::pci_link_is_up` function. This function serves as a simple way to ensure PCI link health. With the introduction of the driver translation API, this function now requires a reference to a BAR which is mapped to CSR space.

The folowing is a code conversion example:

pci_link_is_up conversion

```
//Pre SDK 1.9 code
bool link_up = acxsdk::pci_link_is_up(device.get_device());

//Post SDK 1.9 code
uint32_t csr_bar_id = 3;
bool link_up = acxsdk::pci_link_is_up(device.get_device(), device.get_bar_handle(csr_bar_id));
```

Revision History

Version	Date	Description
1.0	21 Oct 2022	• Initial Achronix release.
1.1	13 Jul 2023	• Added Achronix PCI device driver • Add installation and configurations details for both BittWare and Achronix drivers • Added API for MSI-X interrupts • Added conversion information to v1.9 code base