Software Development Kit User Guide (UG107)

All Achronix Devices

Achronix®

Data Acceleration

Copyrights, Trademarks and Disclaimers

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 9
Software Stack 9
Chapter - 2: Installation 10
Prerequisites 10
Achronix SDK installation 11
Directory Structure 11
Compiling the Achronix SDK 12
Testing the Achronix SDK 12
Chapter - 3: Developing Applications 14
Minimum Requirements 14
Compilation 14
Runtime 14
Chapter - 4: Memory Addressing 15
FPGA Addressing 15
Host Addressing 16
BAR Assignment 16
Device 16
Software 18
Recommendations 18
Address Translation Unit (ATU) 18
Bar Match Mode 20
Address Match Mode 20
DMA Transfers 20
2D NoC Physical Address Calculations 20
Basic Operation 21
Linked List Mode 22
Design Requirements 24
Achronix_DDR4.cpp 24
Achronix_GDDR6.cpp 24
Achronix_PCI.cpp 24

DMA_example.cpp 25

Chapter - 5: SDK Functions 26

Quick Reference Table 26

util_calc_nap_absolute_addr() 27

Description 27

Call 27

Arguments 27

Return Value 27

util_wait_microseconds() 28

Description 28

Call 28

Arguments 28

Return Value 28

util_wait_seconds() 29

Description 29

Call 29

Arguments 29

Return Value 29

pci_reg_write_offset() 30

Description 30

Call 30

Arguments 30

Return Value 30

pci_reg_read_offset() 31

Description 31

Call 31

Arguments 31

Return Value 31

pci_reg_set_bits_offset() 32

Description 32

Call 32

Arguments 32

Return Value 32

pci_reg_clear_bits_offset() 33

Description 33

Call 33

Arguments . . . 33
Return Value . . . 33

pci_read_reg_ctrl_version() . . . 34
Description . . . 34
Call . . . 34
Arguments . . . 34
Return Value . . . 34

pci_link_is_up() . . . 35
Description . . . 35
Call . . . 35
Arguments . . . 35
Return Value . . . 35

dma_build_data_descriptor() . . . 36
Description . . . 36
Call . . . 36
Arguments . . . 36
Return Value . . . 36

dma_build_link_descriptor() . . . 37
Description . . . 37
Call . . . 37
Arguments . . . 37
Return Value . . . 37

dma_init() . . . 38
Description . . . 38
Call . . . 38
Arguments . . . 38
Return Value . . . 38

dma_config() . . . 39
Description . . . 39
Arguments . . . 39
Return Value . . . 39

dma_start() . . . 40
Description . . . 40
Call . . . 40
Arguments . . . 40
Return Value . . . 40

dma_halt() . . . 41
- Description . . . 41
- Call . . . 41
- Arguments . . . 41
- Return Value . . . 41

dma_get_status() . . . 42
- Description . . . 42
- Call . . . 42
- Arguments . . . 42
- Return Value . . . 42

dma_wait() . . . 43
- Description . . . 43
- Call . . . 43
- Arguments . . . 43
- Return Value . . . 43

atu_get_context() . . . 44
- Description . . . 44
- Call . . . 44
- Arguments . . . 44
- Return Value . . . 44

atu_find_regions() . . . 45
- Description . . . 45
- Call . . . 45
- Arguments . . . 45
- Return Value . . . 45

atu_get_region() . . . 46
- Description . . . 46
- Call . . . 46
- Arguments . . . 46
- Return Value . . . 46

atu_put_region() . . . 47
- Description . . . 47
- Call . . . 47
- Arguments . . . 47
- Return Value . . . 47

Chapter - 6: SDK Structures . . . 48

DmaCommand_t 48
Description 48
Definition 48
Fields 48
DMADataDescriptor 50
Description 50
Definition 50
Fields 50
DMALinkDescriptor 51
Description 51
Definition 51
Fields 51
Chapter - 7: SDK Classes 52
PCIDevice 52
Description 52
Definition 52
Member Functions 52
DMAHostBuffer 54
Description 54
Definition 54
Member Functions 54
DMADescriptorList 56
Description 56
Definition 58
Member Functions 58
ATUContext 60
Description 60
Definition 60
Member Functions 60
ATURegion 61
Description 61
Definition 61
Member Functions 62
Revision History 65

Chapter - 1: Introduction

The Achronix Software Development Kit (SDK) is a set of functions and data structures which enable users of the Speedster®7t family of devices 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.

Note

The Achronix SDK is currently only available for the Linux platform with an installed Achronix VectorPath® S7t-VG6 accelerator card. For Microsoft Windows, or non-VectorPath support, please contact support@achronix.com.

Software Stack

The Achronix SDK is built on top of the BittWare SDK. The BittWare SDK supplies the low-level routines performing PCIe device enumeration and recognition, buffer allocation, and memory-mapped reads and writes via the PCIe device.

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

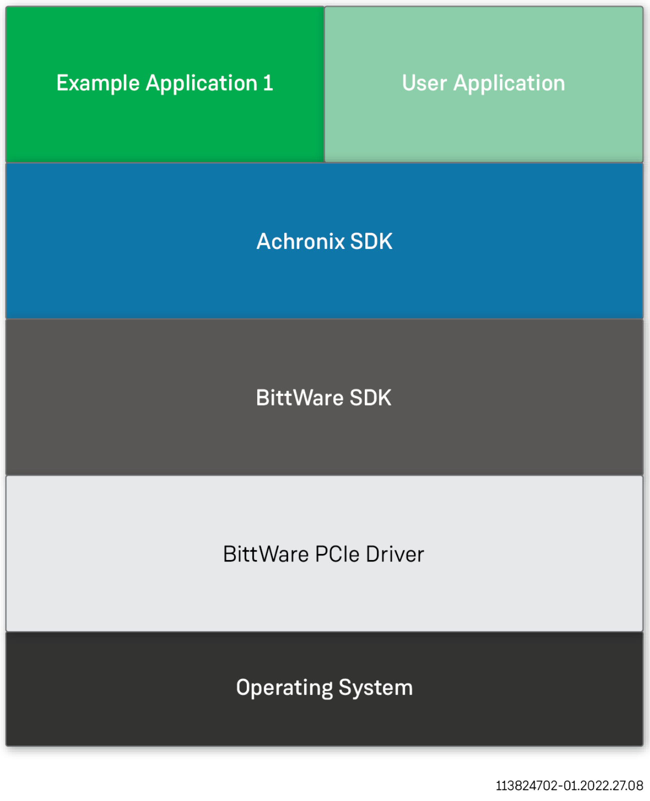


Figure 1: *Software Stack*

Chapter - 2: Installation

Prerequisites

Before compiling code using the Achronix SDK, install the BittWare SDK available from the BittWare VectorPath developer portal at https://developer.bittware.com. See the Support Article for help getting started. Access to the portal and SDK download is available only to verified purchasers of the VectorPath card. Please contact BittWare Support to obtain a developer account.

The default locations for installation of the BittWare SDK are shown below:

```
BittWare SDK FIle Locations

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

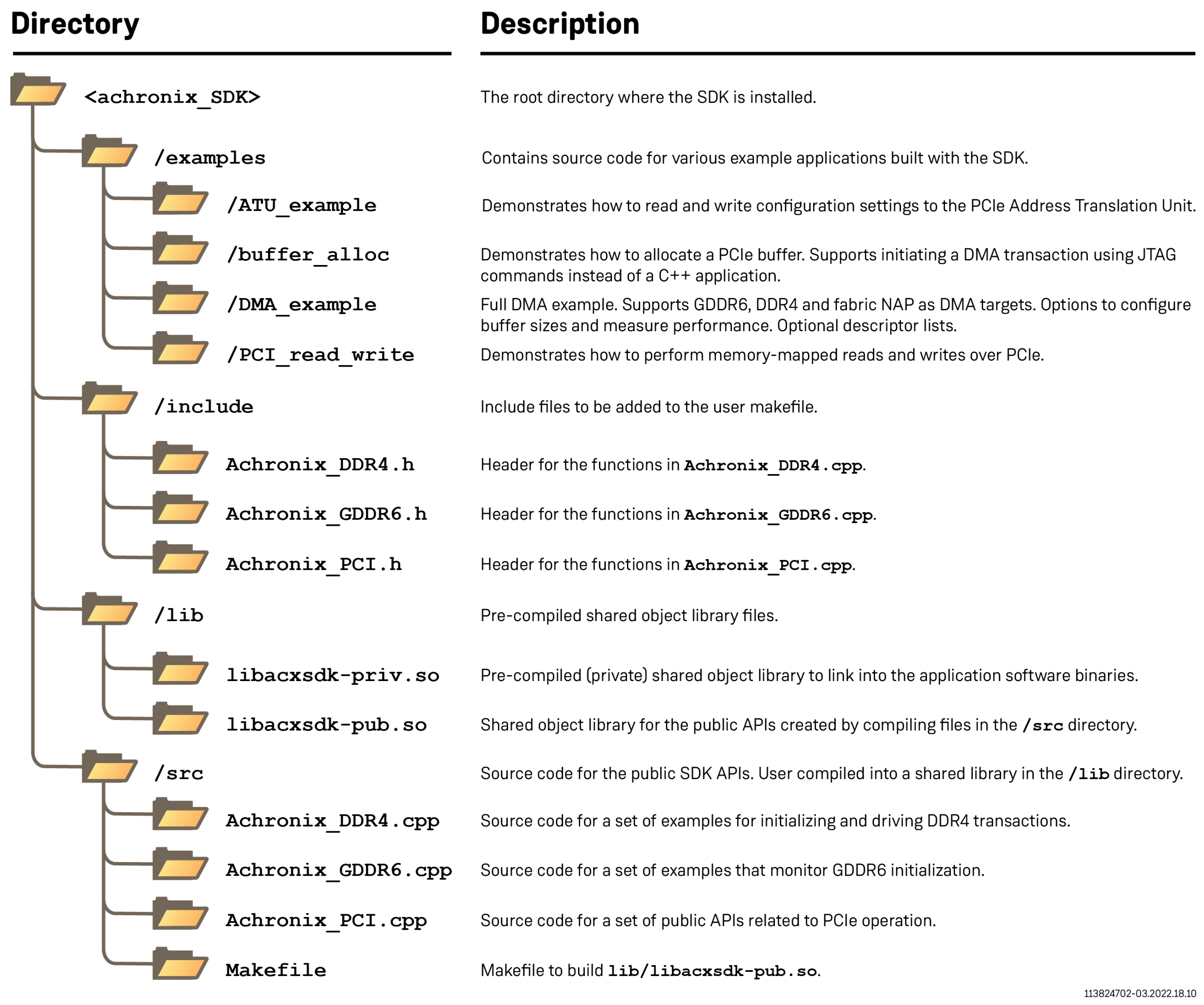
If the BittWare SDK is installed to a different location, the appropriate entries in each of the Achronix makefiles must be updated.

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.

Achronix SDK installation

Directory Structure

The Achronix SDK has the following directory structure:



Directory	Description
`<achronix_SDK>`	The root directory where the SDK is installed.
`/examples`	Contains source code for various example applications built with the SDK.
`/ATU_example`	Demonstrates how to read and write configuration settings to the PCIe Address Translation Unit.
`/buffer_alloc`	Demonstrates how to allocate a PCIe buffer. Supports initiating a DMA transaction using JTAG commands instead of a C++ application.
`/DMA_example`	Full DMA example. Supports GDDR6, DDR4 and fabric NAP as DMA targets. Options to configure buffer sizes and measure performance. Optional descriptor lists.
`/PCI_read_write`	Demonstrates how to perform memory-mapped reads and writes over PCIe.
`/include`	Include files to be added to the user makefile.
`Achronix_DDR4.h`	Header for the functions in **`Achronix_DDR4.cpp`**.
`Achronix_GDDR6.h`	Header for the functions in **`Achronix_GDDR6.cpp`**.
`Achronix_PCI.h`	Header for the functions in **`Achronix_PCI.cpp`**.
`/lib`	Pre-compiled shared object library files.
`libacxsdk-priv.so`	Pre-compiled (private) shared object library to link into the application software binaries.
`libacxsdk-pub.so`	Shared object library for the public APIs created by compiling files in the **`/src`** directory.
`/src`	Source code for the public SDK APIs. User compiled into a shared library in the **`/lib`** directory.
`Achronix_DDR4.cpp`	Source code for a set of examples for initializing and driving DDR4 transactions.
`Achronix_GDDR6.cpp`	Source code for a set of examples that monitor GDDR6 initialization.
`Achronix_PCI.cpp`	Source code for a set of public APIs related to PCIe operation.
`Makefile`	Makefile to build **`lib/libacxsdk-pub.so`**.

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

The Achronix SDK may be copied and installed to any location that best suits the user application. When installed, it is necessary to set the `LD_LIBRARY_PATH` environment variable to include the `<achronix_SDK>/lib` directory.

Compiling the Achronix SDK

Compiling the Achronix SDK is optional and only required if the source code has been modified. Pre-compiled versions of the shared libraries and the demo application executables are included with the SDK. Before compiling an application using the Achronix SDK, it is necessary to compile the `libsdk-pub.so` shared object file as shown below:

```
# To compile the libsdk-pub shared object file
$ cd <achronix_SDK>/src     -  Change to the source code directory.
$ make clean               -  Delete all object (.o) and /lib/libacxsdk-pub.so files.
$ make                     -  Compile /lib/libacxsdk-pub.so.
```

The Achronix SDK libraries (`libacxsdk-priv.so` and `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.

It is then necessary to compile the supplied example applications. Instructions for compiling the DMA_example are shown below. These steps can be repeated for the other example applications.

```
# To compile the DMA_example application
$ cd <achronix_SDK>/examples/DMA_example -  Change to the application directory.
$ make clean                             -  Delete all object (.o) and any executable files.
$ make                                   -  Compile the application.
```

This builds an executable named `<achronix_SDK>/sw/examples/dma_example` that can be used as a model for developing your own applications.

Note

When the application is compiled, it might be necessary to set execute permissions on the file. This is achieved with the command: `chmod +x <application>`.

Testing the Achronix SDK

To verify that the software and the VectorPath card are installed properly, run the DMA example as follows:

1. Run the following command to see the available command line options:

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

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

```
$ <achronix_SDK>/sw/examples/dma_example -b 0x400000 -d H2D2H -e DDR4 -f random
```

The test does the following:

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

Chapter - 3: Developing Applications

To develop your own 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 BittWare and Achronix shared library files. Consult the included Makefiles for more detail.

Minimum Requirements

Compilation

The following are the minimum requirements necessary to compile the Achronix SDK into user software:

- The environment variable `LD_LIBRARY_PATH` must include `<achronix_SDK>/lib`
- Include the Achronix SDK header files in the `<achronix_SDK>/include` directory into C/C++ code
- Include the Achronix SDK shared object files in the `<achronix_SDK>/lib` directory when linking

Runtime

- Ensure that the BittWare library files are installed in `/usr/lib/x86_64-linux-gnu`

Chapter - 4: Memory Addressing

In order to determine the correct address for an FPGA memory space, whether within the interface subsystems or the fabric, it is necessary to understand the differences between how the FPGA and the software map the address space.

FPGA Addressing

The FPGA maps all addresses as a 42-bit linear address space as described in the *Speedster7t Network on Chip User Guide* (UG089). In this linear space, each interface subsystem can be address 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 and within the device. However, they are addressed from 0.
2. The function `util_calc_nap_absolute_address()` returns bits [41:28] of the NAP address when given the NAP column and row locations (numbered from 1).

In summary, the FPGA memory space is a 42-bit linear address space giving access to all registers and external memory locations.

Host Addressing

PCIe devices such as the VectorPath card are memory-mapped, 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. Reads and writes to registers or memories on the FPGA are performed by reading or writing to the appropriate address on the host.

In order to implement this memory-mapping scheme, PCIe uses BARs (Base Address Registers) to specify the mapping from the host machine address space to the PCIe device address space(s). Each BAR defines the size of an address space and the local (device) base address. During PCIe enumeration, the host assigns a memory region of the same size as each BAR, and maps this to the host (usually 64-bit) physical memory space. The local, or device, base address is the start of a memory region of the same size in the FPGA 42-bit address space (as described in FPGA Addressing (see page 15)). When host software reads from, or writes to, a physical address covered by one of the BARs, that transaction is intercepted by the host PCIe controller and routed to the correct location on the FPGA.

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. It is therefore clear that the 42-bit address space within the device cannot be mapped to the host in its entirety. It is necessary to configure the BARs and their sizes to match the individual user design requirements. The size of, and FPGA base addresses assigned to, each BAR are defined in the PCIe IP configurator. Refer to the *Speedster7t PCIe User Guide* (UG098), available under NDA. The base addresses in the host address space are assigned by the host during PCIe device enumeration. For more information about how to access the BAR registers from the host, refer to the BittWare SDK User Guide.

It is important to understand that BAR register configuration is specific to a design. The capabilities of a user application must match the design BAR register selection. When selecting CSR register spaces, or the NAP to address, the design and user application BAR definitions must be aligned.

Warning

The BAR register selection is specific to a design. The user application must match the BAR register selection.

Note

DMA accesses do not use BAR registers. DMA transfers use the physical address within the device and host.

BAR Assignment

To understand the alignment between the design and the user application BAR assignment, the combined demonstration design (`pcie_gddr6_ddr4_vp_demo`) is used as an example. This 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.

Device

The PCIe IP definitions, including the BAR assignments, are specified in the ACE I/O Designer, which is accessed from the IP Configuration Perspective in the ACE GUI. The BAR assignments, along with other PCIe configurations for the above demonstration design, are contained within `/src/acxip/pcie_express_x16.acxip`. The BARs are configured in the I/O Designer as shown in the following table.

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

BAR	Type	Size (Bytes)	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 MSIX interrupts.
3	Memory	1M	`0x081_9100_0000`	Base of `CSR_SPACE PCIE_1 BASE_IP`.
4	Memory	1M	`0x002_0000_0000`	GDDR 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 `/src/constraints/ace_placements.pdc`, it can be seen that this mapping relates to the fabric-based register control block. This block controls the design, operating the DDR4 training and transactions, and monitoring the GDDR6 training status:

```
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 3 is mapped to the Control and Status Register (CSR) space within the PCIe core. This mapping is required so the software can 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 core register space (address 0x081_9100_0000). The number of that BAR must then be used in the application to access any DMA or PCIe core functions.

Software

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 core registers:

```
Excerpt From DMA_example.cpp

// Configure mapping for each BAR.
// IMPORTANT - These can change on a per-design basis.
BwpciMs reg_ctrl_bar = BW_MS_BAR0;  // Mapped to the register control block in the fabric
BwpciMs csr_bar      = BW_MS_BAR3;  // Mapped to the Control Status Registers

....


// DDR4 training is controlled and monitored by the reg_ctrl block, which will be
// accessed by one of the BARs
if ( acx::ddr4_run_training( device, reg_ctrl_bar, false) != 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 );
```

Recommendations

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

- Assign a BAR to the PCIe core registers (`0x081_9100_0000`) and use this BAR for all PCIe core accesses in the software.
- If the design contains a register control block with a NAP, define a BAR with an address that matches the location defined in the `ace_placements.pdc` file. In the software use this BAR for all accesses to the register control block.
- DMA transfers do not use BARs. To obtain the absolute address of a NAP for DMA access, use `acx::util_calc_nap_absolute_address()`. For GDDR6 or DDR4 absolute 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 access. For example, see BAR1, BAR4, and BAR5 defined in the table above.

Address Translation Unit (ATU)

Mapping between the host 64-bit physical address space, and the device 42-bit NoC address space, is performed by the Address Translation Unit (ATU) part of the PCIe interface subsystem. When a user design accesses an address that has been assigned by the host OS to one of the device BARs, that address is passed to the ATU for translation from host address to device NoC address. If the ATU is 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 NoC without translation. This would create an invalid address which would cause NoC access failures. The ATU is configured, using ACE, whenever a PCIe IP is configured and assigned BARs with an address offset and size.

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 on 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.

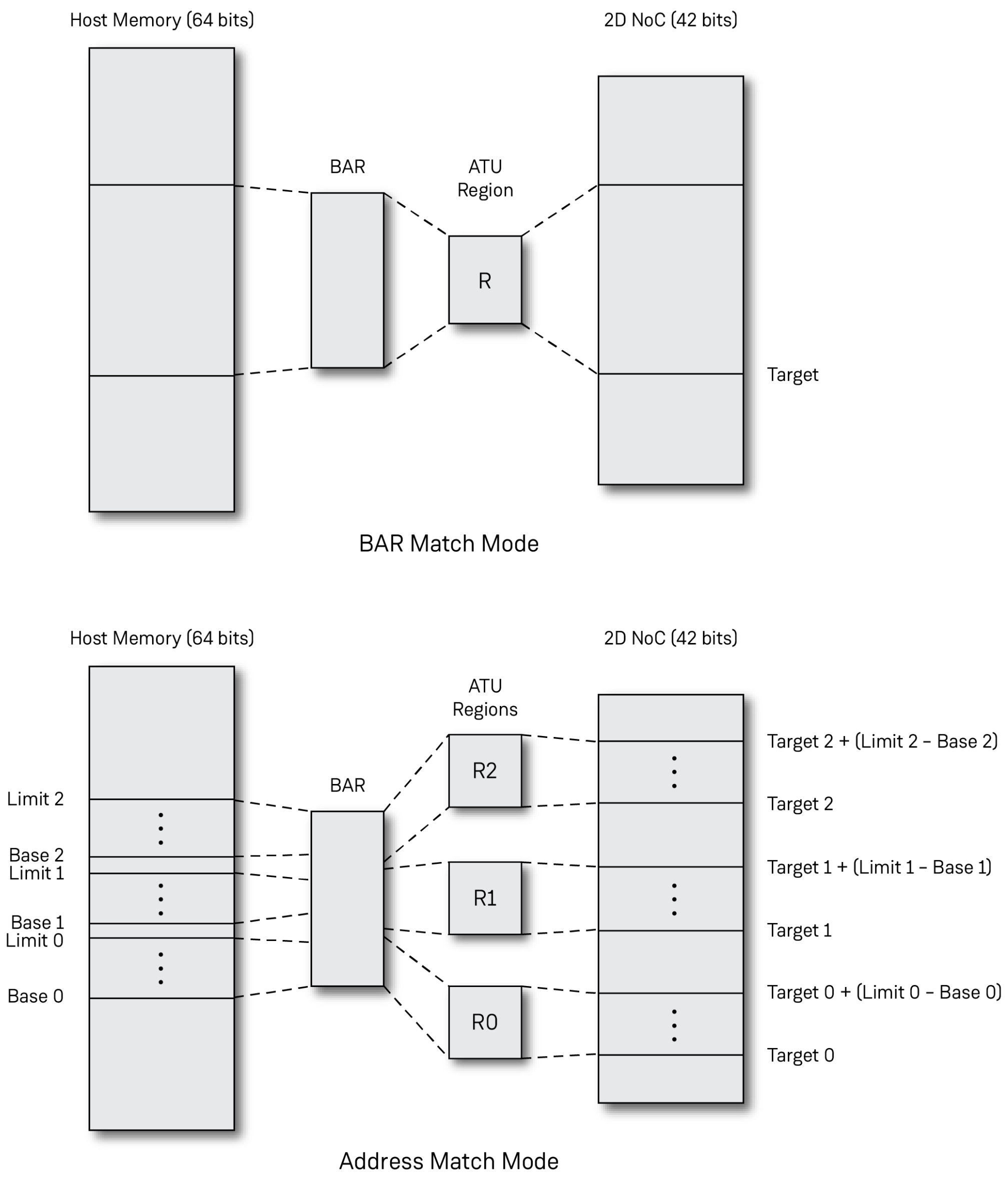


Figure 3: *ATU Region Examples*

Bar Match Mode

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 there are only six 32-bit BARs, a maximum of six ATU regions can be configured in Bar Match Mode. In this mode, only the device-side Target address for the BAR must be specified by the user, 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 a base address in host memory, a base address in device memory, and 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 device memory. 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.

Note

Currently the ACE I/O Designer tool only supports BAR Match Mode. Address Match Mode can only be configured by using the Achronix SDK capabilities described below.

DMA Transfers

Direct Memory Access (DMA) transfers do 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 device *local* 42-bit address space. 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.

2D NoC Physical Address Calculations

Note

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

To obtain the 42-bit 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 GDDR6 channel (two per controller) is addressed with `addr[36:33]`. It is therefore necessary to add `0x2_0000_0000` for each GDDR6 channel selected.

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) {
    // Leave room for descriptor lists at the base of DDR4.  The DDR4 space starts from offset
0x0.
    device_phys_base_addr = DDR4_SPACE + (uint64_t)0x10000;
} 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 Operation

In order to perform a DMA transfer, the software must first open the PCIe device and obtain a handle to it. With the Achronix SDK, this is achieved using the underlying BittWare SDK functions. Alternately, this handle can be obtained directly by using low-level OS system calls (out of the scope of this document). 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 BittWare `bw_card_list` utility is used to determine the appropriate `device_id`. 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 host-to-device DMA, copy the host data to the host buffer. The Achronix SDK utilizes the BittWare SDK functions to allocate that buffer providing a C++ class named `acxsdk::DMAHostBuffer`. Again, if necessary, it is possible to use direct low-level operating system calls. Using `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.

Warning

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

The example below 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 may be edited 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 `/src/Achronix_PCI.cpp`; They can be customized if required.

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

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:

- `BwpciDevice` pointer obtained by calling `acxsdk::DMAHostBuffer::get_device()`
- `BwpciMs` object for a BAR that maps to the CSR space in the FPGA
- Defines for the desired Achronix part name and PCIe controller number.

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

Next, 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:

1. The transfer direction.
2. The 42-bit `device_address` (calculated in the code example above).
3. The 64-bit `host_address`, obtained by calling `DMAHostBuffer::get_phys_addr()`.
4. 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;
```

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

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_second
s*/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 described above, the more advanced linked list mode is available to handle larger DMA transfers or streaming transactions more efficiently. 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 my 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 example below, 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.

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`.

```
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));
}
```

After building the `DMADescriptorList`, the list must be transferred into device memory, which can be performed using the DMA Basic Operation (see page 21) procedure outlined above. 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 example above, the physical address of token `GDDR6_SPACE` is used. This address equates to the lowest of GDDR6 memory addresses in the 42-bit NoC address space. For more information, see the `DMA_example` source code.

After the descriptor list is complete, the `DmaCommand` structure is populated with the DMA transfer parameters. Comparing the linked list mode example below with the Basic Operation (see page 21) example above, 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();
```

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.

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 functions above 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 - 5: 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 is shown below.

```
// 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        (BwpciDevice *device, BwpciMs csr_bar, uint32_t addr_offset,
uint32_t value );
uint32_t pci_reg_read_offset         (BwpciDevice *device, BwpciMs csr_bar, uint32_t addr_offset
);
int      pci_reg_set_bits_offset     (BwpciDevice *device, BwpciMs csr_bar, uint32_t addr_offset,
int start_bit, int stop_bit );
int      pci_reg_clear_bits_offset   (BwpciDevice *device, BwpciMs csr_bar, uint32_t addr_offset,
int start_bit, int stop_bit );

// PCI specific functions
void     pci_read_reg_ctrl_version   (BwpciDevice *device, BwpciMs reg_ctrl_bar);
bool     pci_link_is_up              (BwpciDevice *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                   (BwpciDevice *device, BwpciMs csr_bar, PartName part,
PCIeCoreNum core);
void      dma_config                 (BwpciDevice *device, DmaCommand_t &p_dma_inst);
void      dma_start                  (BwpciDevice *device, DmaCommand_t &p_dma_inst);
void      dma_halt                   (BwpciDevice *device, DmaCommand_t &p_dma_inst);
DmaStatus dma_get_status             (BwpciDevice *device, DmaCommand_t &p_dma_inst);
DmaStatus dma_wait                   (BwpciDevice *device, DmaCommand_t &command, int
timeout_in_seconds);

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

util_calc_nap_absolute_addr()

Description

Calculate the absolute 42-bit address of a NAP, placed in the 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 absolute addresses. 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

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

The function returns the absolute 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

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

Return Value

The function has no 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

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

Return Value

The function returns the absolute NoC 42-bit address of the NAP.

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 (BwpciDevice *device, BwpciMs csr_bar, uint32_t
addr_offset, uint32_t value );
```

Arguments

Type	Argument	Description
`BwpciDevice*`	`device`	Pointer to the PCIe device.
`BwpciMs`	`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

The function 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 (PCIDevice &device, BwpciMs csr_bar, uint32_t
addr_offset);
```

Arguments

Type	Argument	Description
`BwpciDevice*`	`device`	Pointer to the PCIe device.
`BwpciMs`	`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

The function 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 (BwpciDevice *device, BwpciMs csr_bar, uint32_t
addr_offset, int start_bit, int stop_bit);
```

Arguments

Type	Argument	Description
`BWpciDevice*`	`device`	Pointer to the PCIe device.
`BwpciMs`	`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

The function 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 (BwpciDevice *device, BwpciMs csr_bar, uint32_t
addr_offset, int start_bit, int stop_bit);
```

Arguments

Type	Argument	Description
`BwpciDevice*`	`device`	Pointer to the PCIe device.
`BwpciMs`	`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

The function 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

```
void pci_read_reg_ctrl_version (BwpciDevice *device, BwpciMs reg_ctrl_bar);
```

Arguments

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

Return Value

The function returns 0.

pci_link_is_up()

Description

Utility function to indicate if the PCIe device is correctly enumerated and available for access. The function confirms that the Vendor ID register returns the expected value.

Call

```
bool pci_link_is_up (BwpciDevice *device);
```

Arguments

Type	Argument	Description
`BwpciDevice*`	`device`	Pointer to the PCIe device.

Return Value

The function 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

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 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_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

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. Causes DMA to complete. Defined as device absolute physical address within the 42-bit 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 (BwpciDevice *device, BwpciMs csr_bar, PartName part, PCIeCoreNum
core);
```

Arguments

Type	Argument	Description
`BwpciDevice*`	`device`	Pointer to the PCIe device.
`BwpciMs`	`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

The function 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 link-lists are being used, the structure includes the address of the start of the linked list in the device 42-bit NoC address space.

Note

This function does not start a DMA transfer.

Call

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

Arguments

Type	Argument	Description
`BwpciDevice*`	`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 `dma_config()`.

Call

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

Arguments

Type	Argument	Description
`BwpciDevice*`	`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 (BwpciDevice *device, DmaCommand &dma_command);
```

Arguments

Type	Argument	Description
`BwpciDevice*`	`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_get_status()

Description

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

Call

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

Arguments

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

Return Value

The function returns one of the values for DMA status 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 return to the caller when the transfer is complete.

Note

This function is multi-threaded and non-blocking.

Call

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

Arguments

Type	Argument	Description
`BwpciDevice*`	`device`	Pointer to the PCIe device.
`DmaCommand&`	`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

The function 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

```
atu_get_context (BwpciDevice *device, BwpciMs csr_bar, PCIeCoreNum core,
ATUContext &context);
```

Arguments

Type	Argument	Description
`BwpciDevice*`	`device`	Pointer to the PCIe device.
`BwpciMs`	`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

```
atu_find_region (BwpciDevice *device, BwpciMs csr_bar, PCIeCoreNum core, int
bar_num, std::vector<ATURegion> &regions);
```

Arguments

Type	Argument	Description
`BwpciDevice*`	`device`	Pointer to the PCIe device.
`BwpciMs`	`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`.
`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

```
atu_get_region (BwpciDevice *device, BwpciMs csr_bar, PCIeCoreNum pcie_core, int
region_num, ATURegion &region);
```

Arguments

Type	Argument	Description
`BwpciDevice*`	`device`	Pointer to the PCIe device.
`BwpciMs`	`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

```
atu_put_region (BwpciDevice *device, BwpciMs csr_bar, PCIeCoreNum pcie_core,
ATURegion &region);;
```

Arguments

Type	Argument	Description
`BwpciDevice*`	`device`	Pointer to the PCIe device.
`BwpciMs`	`csr_bar`	BAR that references the Configuration Status Registers in the PCIe DBI space.
`PCIeCoreNum`	`region_num`	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.

Chapter - 6: 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 {
    BwpciMs          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

Type	Parameter	Description
`BwpciMs`	`csr_bar`	A handle to a BAR register mapped to the base of CSR space in the 42-bit 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).
`uint64_t`	`device_address`	The 42-bit base address for the transfer on the device side (normal mode only).

Type	Parameter	Description
`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 amount of verbose 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 below.

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

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 below), 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

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 - 7: SDK Classes

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

PCIDevice

Description

This is a convenience class that is a C++ wrapper around the `BwpciDevice` and associated API from the BittWare toolkit. When called, the constructor attempts to open the PCIe device with the specified `device_id`. When the 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. The `device_id` can be obtained using the `bw_card_list` command.

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; }
    BwpciDevice *get_device() { return _device; }
    void print();
    // PCI Reads
    int read_uint8(BwpciMs memory_space, uint64_t offset, uint8_t *buffer, int count);
    int read_uint16(BwpciMs memory_space, uint64_t offset, uint16_t *buffer, int count);
    int read_uint32(BwpciMs memory_space, uint64_t offset, uint32_t *buffer, int count);
    int read_uint64(BwpciMs memory_space, uint64_t offset, uint64_t *buffer, int count);
    // PCI Writes
    int write_uint8(BwpciMs memory_space, uint64_t offset, uint8_t *buffer, int count);
    int write_uint16(BwpciMs memory_space, uint64_t offset, uint16_t *buffer, int count);
    int write_uint32(BwpciMs memory_space, uint64_t offset, uint32_t *buffer, int count);
    int write_uint64(BwpciMs memory_space, uint64_t offset, uint64_t *buffer, int count);
};
```

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 that can be obtained with the `get_device()` function.

Return Type	Function	Description
`void`	`~PCIDevice`	Descructor. Closes the PCI device if it is open.
`DeviceStatus`	`get_pci_status`	Returns an enum value indicating whether the device was opened successfully or not.
`int`	`get_device_id`	Returns the `device_id` passed into the constructor.
`BwpciDevice`	`get_device`	Returns a handle to the underlying BittWare `BwpciDevice` object if 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`	A wrapper around the `bwpci_ms_read8s` function callable through the class.
`int`	`read_uint16`	A wrapper around the `bwpci_ms_read16s` function callable through the class.
`int`	`read_uint32`	A wrapper around the `bwpci_ms_read32s` function callable through the class.
`int`	`read_uint64`	A wrapper around the `bwpci_ms_read64s` function callable through the class.
`int`	`write_uint8`	A wrapper around the `bwpci_ms_write8s` function callable through the class.
`int`	`write_uint16`	A wrapper around the `bwpci_ms_write16s` function callable through the class.
`int`	`write_uint32`	A wrapper around the `bwpci_ms_writ32s` function callable through the class.
`int`	`write_uint64`	A wrapper around the `bwpci_ms_writ64s` function callable through the class.

DMAHostBuffer

Description

A convenience class consisting of a C++ wrapper around the `bwpci_alloc_mem()` function in the BittWare SDK. When called, the constructor allocates a DMA buffer of the given size. When the 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 the buffer and to fill it 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

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 whether the buffer was allocated successfully or not.
`uint64_t`	`get_size_in_bytes`	Currently, buffers larger than 4MB are not supported.
`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.

Return Type	Function	Description
`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 with 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 figure below. 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 DMA can begin. They 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.

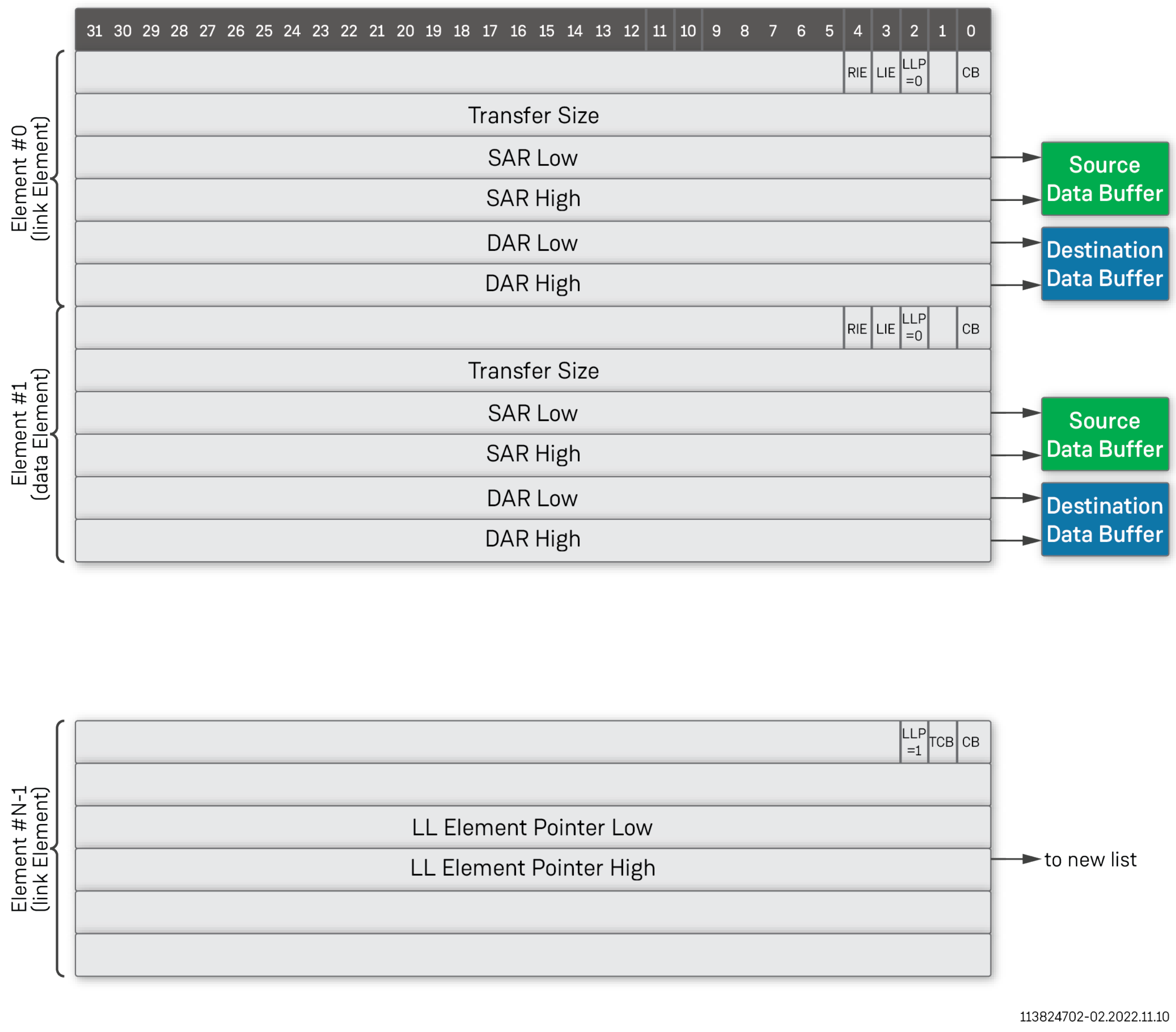


Figure 4: *DMADescriptorList Structure in Memory*

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

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 whether the descriptor list was allocated successfully or not.
`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 your 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 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[]`	This 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 below).

Definition

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

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

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.
`bool`	`get_single_addr_loc_trans_en`	Reserved for future use.

Return Type	Function	Description
`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 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`.
`void`	`set_invert_mode`	Reserved for future use.

Return Type	Function	Description
`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 NoC address space. The device region size must be the same as on the host, so the device limit address is calculated automatically.

Revision History

Version	Date	Description
1.0	21 Oct 2022	• Initial Achronix release.