Introduction

The Xilinx PCI Express DMA (XDMA) IP provides high performance Scatter Gather (SG) direct memory access (DMA) via PCI Express. Using the IP and the associated drivers and software one will be able to generate high throughput PCIe memory transactions between a host PC and a Xilinx FPGA.

This document provides tips and techniques for debugging XDMA IP issues. As an introduction, an overview of the XDMA architecture is provided along with its working mechanism. For more details, users are advised to check XDMA IP product guide (PG195).

At the end of this document, the details on how the XDMA IP legacy drivers, provided in (Xilinx Answer 65444), work has been described. The section has been introduced to provide users with the knowledge of the working mechanism of the drivers. If an advanced debugging is required, it is advised to add printf statements at different points in the provided driver source to narrow down the source of the issue.

DMA Architecture and Overview

The XDMA IP consists of the following interfaces as shown in Figure 1:

- **User Data Interface**
  - AXI-MM (Memory Mapped) or AXI-ST (Streaming)
    - Separate data port per channel in AXI-ST Interface; data port is shared between channels in the AXI-MM interface
    - Up to 4 physical Read (H2C) and 4 Write (C2H) Data Channels
      - Each channel enabled has a dedicated engine for H2C and C2H
      - Descriptor module is common for all engines

- **Control Interfaces**
  - AXI-MM Lite Master Control Interface
  - AXI-MM Lite Slave Control Interface accessible from user application

- **DMA Bypass Interface**
  - AXI-MM Bypass Port
    - Enables Host direct access to user application

- **User Interrupts**
  - Up to 16 user interrupts
**Status ports**

- Each channel has a status port

**AXI MM interface**

As shown in Figure 2, the AXI MM data port is shared among the configured channels. C2H channels will master reads on the AR bus and H2C channels will master writes on the AW bus.
AXI Stream Interface

When the IP is configured with the AXI Stream Interface option, each channel will have its own AXI Stream interface as shown in Figure 3.

**Figure 3 - XDMA AXI Stream Interface**

**Descriptor Format**

Table 1 shows descriptor formats. Descriptors reside in the host memory. Each descriptor has a source address, destination address, length, and a pointer to the next descriptor on the list unless the STOP bit is set. The Next_adjacent field indicates how many contiguous descriptors are in the next descriptor address.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Bit Index</th>
<th>Sub Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Magic</td>
<td>15:0</td>
<td></td>
<td>16'had4b. Code to verify that descriptor is valid.</td>
</tr>
<tr>
<td></td>
<td>Nxt_adj</td>
<td>5:0</td>
<td></td>
<td>The number of additional adjacent descriptors after the descriptor located at the next descriptor address field. A block of adjacent descriptor cannot cross a 4K boundary.</td>
</tr>
<tr>
<td>0x04</td>
<td>Control</td>
<td>4</td>
<td>EOP</td>
<td>End of packet (AXI ST C2H only)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Completed</td>
<td>Set to (1) to interrupt after the engine has completed this descriptor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>Stop</td>
<td>Set to (1) to stop the engine when it completes this descriptor.</td>
</tr>
<tr>
<td>0x08</td>
<td>Len</td>
<td>[27:0]</td>
<td></td>
<td>Descriptor Data length</td>
</tr>
<tr>
<td>0x08 &amp; 0x0C</td>
<td>Source Address</td>
<td>63:0</td>
<td></td>
<td>Source address for the DMA transfer</td>
</tr>
<tr>
<td>0x10 &amp; 0x14</td>
<td>Destination Address</td>
<td>63:0</td>
<td></td>
<td>Destination address for the DMA transfer</td>
</tr>
<tr>
<td>0x18 &amp; 0x1C</td>
<td>Next Descriptor Address</td>
<td>63:0</td>
<td></td>
<td>Address of the next descriptor in the list</td>
</tr>
</tbody>
</table>
XDMA BAR Routing

All of the requests from the host will be directed to different interfaces based on the BAR hit. Which interface corresponds to which BAR is shown in Table 2 and Table 3. PCIe to DMA interface is always selected by default. Figure 4 shows the routing mechanism for the incoming requests from the host when “PCIe to AXI Lite Master” and “PCIe to DMA Bypass interfaces” are enabled.

<table>
<thead>
<tr>
<th>Default</th>
<th>BAR0 (32-bit)</th>
<th>BAR1 (32-bit)</th>
<th>BAR2 (32-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIe to AXI Lite Master enabled</td>
<td>PCIe to AXI Lite Master</td>
<td>DMA</td>
<td></td>
</tr>
<tr>
<td>PCIe to AXI Lite Master and PCIe to DMA Bypass enabled</td>
<td>PCIe to AXI Lite Master</td>
<td>DMA</td>
<td>PCIe to DMA Bypass</td>
</tr>
<tr>
<td>PCIe to DMA Bypass enabled</td>
<td>DMA</td>
<td>DMA</td>
<td>PCIe to DMA Bypass</td>
</tr>
</tbody>
</table>

Table 2 - XDMA BAR Routing (32-bit) [Ref: PG195]

<table>
<thead>
<tr>
<th>Default</th>
<th>BAR0 (64-bit)</th>
<th>BAR2 (64-bit)</th>
<th>BAR4 (64-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIe to AXI Lite Master enabled</td>
<td>PCIe to AXI Lite Master</td>
<td>DMA</td>
<td></td>
</tr>
<tr>
<td>PCIe to AXI Lite Master and PCIe to DMA Bypass enabled</td>
<td>PCIe to AXI Lite Master</td>
<td>DMA</td>
<td>PCIe to DMA Bypass</td>
</tr>
<tr>
<td>PCIe to DMA Bypass enabled</td>
<td>DMA</td>
<td>DMA</td>
<td>PCIe to DMA Bypass</td>
</tr>
</tbody>
</table>

Table 3 - XDMA BAR Routing (64-bit) [Ref: PG195]
DMA Driver

The purpose of a DMA driver that sits in the host CPU is to prepare any peripheral DMA transfers, because only the operating system (OS) has full control over the memory system, the file system and the user space processes. First, the peripheral device’s DMA engine is programmed with the source and destination addresses of the memory ranges to copy. Second, the device is signaled to begin the DMA transfer and when the transfer is finished, usually, the device raises interrupts to inform the CPU about transfers that have finished. For each interrupt, an interrupt handler, previously installed by the driver, is called and the finished transfer can be acknowledged accordingly by the OS.

XDMA Linux Driver and Example Application

The XDMA driver provided in (Xilinx Answer 65444) consists of the following user accessible devices. The driver is provided as a reference. It is the user’s responsibility to modify the driver to add specific requirements, or build one from scratch, as per the need of their custom design.

- xdma0_control : to access XDMA registers
- xdma0_user : to access AXI-Lite Master interface
- xdma0_bypass : to access DMA-Bypass interface
- xdma0_h2c_0/1/2/3, xdma0_c2h_0/1/2/3 : to access each channel

There are three tests included in (Xilinx Answer 65444) which are as follows:

- run_test.sh : Script to do basic transfer
  - Will load driver, find out if the design is AXI-MM or AXI_ST and see how many channels are enabled.
  - Will do basic transfer to all enabled channels.
- Check for data integrity
- Report pass or fail

- load_driver.sh: loads driver
- perform_hwcount.sh: for hardware performance

**Example Application**

*(Xilinx Answer 65444)* provides the following applications:

- **dma_to_device**
  - [AXI-MM] `dma_to_device -d /dev/xdma0_h2c_0 -f infile.bin -s 4096 -a 1000 -o 24`
  - [AXI-ST] `dma_to_device -d /dev/xdma0_h2c_0 -f infile.bin -s 4096 -o 24`

- **dma_from_device**
  - [AXI-MM] `dma_from_device -d /dev/xdma0_c2h_0 -f outfile.bin -s 4096 -a 1000 -o 24`
  - [AXI-ST] `dma_from_device -d /dev/xdma0_c2h_0 -f outfile.bin -s 4096 -o 24`

- **reg_rw**

  - Linux utility ‘dd’ can also be used for the DMA. Linux ‘dd’ is a basic Linux utility to copy. It also gives bandwidth information
    - `dd if=/dev/zero of=/dev/xdma0_h2c_0 bs=4096 count=1`  
      - Will transfer 4Kbytes from Host to Card
    - `dd of=/dev/null if=/dev/xdma0_c2h_0 bs=4096 count=1`  
      - Will transfer 4Kbytes from Card to Host

DMA Transfer flow for H2C and C2H

Register Programming during ‘driver load’ process

```
Load driver (setup)

Set ‘H2C Channel interrupt enable mask’ register 0x0090 (Table 2-49) to generate interrupts for corresponding bits.

Set ‘C2H Channel interrupt enable mask’ register 0x1090 (Table 2-68) to generate interrupts for corresponding bits.

Set ‘IRQ Block Channel Interrupt Enable Mask’ register 0x2010 (Table 2-81), enable all channels both H2C and C2H to generate interrupt.
```

© Copyright 2018 Xilinx
H2C Transfer

Application program initiates H2C transfer, with transfer length, buffer location where data is stored

Driver creates descriptors based on transfer length

Driver writes first descriptor base address to Address 0x4080 (Table 2-108) and 0x4084 (Table 2-109). Driver writes next adjacent descriptor count to 0x4088 (Table 2-110) if any.

Driver starts H2C transfer by writing to H2C engines control register, address 0x0004 (Table 2-40)

DMA initiates Descriptor fetch request for one or more descriptors (depending on adjacent descriptor count)

DMA receives one Descriptor or more descriptors (depending on adjacent descriptor count)

No

Is this the last descriptor

Yes

DMA sends read request to (Host) source address based on first available descriptor

DMA receives data from Host for that descriptor

Stop fetching data from Host

Yes

Transmit data on (Card) AXI-MM Master interface

Is there more data to transfer

No

Send interrupt to Host

Interrupt process. Read 'IRQ Block Channel Interrupt Request' 0x2044 (Table 2-85) to see which channels sent interrupt. Mask corresponding channel interrupt writing to 0x2018 (Table 2-83)

Driver Reads corresponding 'Status register' 0x0044 (Table 2-44) which will also clear status register. Read channel 'Completed descriptor count' 0x0048 (Table 2-45) and compare with number of descriptor generated.

Write to channel 'Control register' 0x0004 (Table 2-40) to stop dma run. Write to 'Block channel interrupt Enable Mask' 0x2014 (Table 2-82) to enable interrupt for next transfer. Return control to application program with transfer size

Yes

Exit application program

© Copyright 2018 Xilinx
In Figure 6, the actions performed by the XDMA driver (shown in yellow boxes) are also reflected in the dmesg log.

The snapshots below are excerpts from the dmesg log taken by running the test application (run_test.sh) that comes with the (Xilinx Answer 65444) driver.
Note: Some driver tasks, as explained in the flow chart, such as reading the completed descriptor count register and writing to the Block channel interrupt enable Mask register, are not explicitly visible in the dmesg log.
C2H Transfer

Figure 7 - C2H Transfer Flowchart [Ref: PG195]

Similar to the H2C transfer, below are excerpts from the dmesg log of a C2H transfer obtained by running the example application (run_test.sh).

Application program initiates C2H transfer, with transfer length, receive buffer location

Driver creates descriptors based on transfer length

Driver writes first descriptor base address to Address 0x5080 (Table 2-114) and 0x5084 (Table 2-115). Driver writes next adjacent descriptor count to 0x5088 (Table 2-116) if any

Driver starts C2H transfer by writing to C2H engines control register, address 0x1004 (Table 2-59)

DMA initiates Descriptor fetch request for one or more descriptors (depending on adjacent descriptor count)

DMA receives one Descriptor or more descriptors (depending on adjacent descriptor count)

DMA reads data from (Card) Source address for a given descriptor

Is this the last descriptor

Yes

Stop fetching descriptor from host

No

Is there any more descriptor left

Yes

Stop fetching data from Card

No

Is there more data to transfer

Yes

Send interrupt to Host

No

Transmit data to PCIe to (Host) Destination address

Interrupt process. Read 'IRQ Block Channel Interrupt Request' 0x2044 (Table 2-85) to see which channels sent interrupt

Mask corresponding channel interrupt writing to 0x2018 (Table 2-83)

Driver Reads corresponding 'Status register' 0x1044 (Table 2-63) which will also clear status register

Read channel 'completed descriptor count' 0x1048 (Table 2-64) and compare with number of descriptor generated.

Write to channel 'Control register' 0x1004 (Table 2-59) to stop dma run.
Write to 'Block channel interrupt Enable Mask' 0x2014 (Table 2-82) to enable interrupt for next transfer

Return control to application program with transfer size

Application program reads transfer data from assigned buffer and writes to a file

Exit application program

© Copyright 2018 Xilinx
Driver creates descriptors based on transfer length.

Driver writes first descriptor base address to Address 0x5060 and 0x5064 Driver writes next adjacent descriptor count to 0x5088 if any.

Driver starts C2H transfer by writing to C2H engines control register, address 0x1004.

Receives interrupt from DMA engine after transfer completion and services it.

Driver Reads corresponding ‘Status register’ 0x1044 which will also clears status register.

Write to channel ‘Control register’ 0x1004 to stop the DMA run.
Other Driver Options

- **Poll mode**
  - No interrupts are used
  - `insmod ../driver/xdma.ko poll_mode=1`

- **Descriptor Credit based transfer**
  - Handshake between Software and Hardware
    - Software gives descriptor credits for hardware to use
    - once hardware finishes credits, it will wait for more credits
  - In the current XDMA driver, ‘Descriptor Credit based transfer’ is supported only for C2H Streaming
  - `insmod ../driver/xdma.ko enable_credit_mp=1`

**Windows Driver**

Windows driver concept is same as in Linux driver. Debug messages can be traced using Trace View program which is part of WDK. See the Windows Driver document in [Xilinx Answer 65444](https://www.xilinx.com) for details.

**lspci**

Figure 8 shows a sample lspci output log. Lspci is helpful in preliminary debug of XDMA in the Linux environment. The lspci log provides the following useful information pertaining to the XDMA operation.

- **BAR information**: One can check and confirm the BAR addresses that are assigned and the size of each BAR.
- **Link Status**: Shows the link status to reflect the actual trained link speed. This should be checked first if low throughput performance is observed than what is expected.
- **Interrupts**: The interrupts being used: Legacy, MSI, MSI-X.
- **Errors**: Gives details on all the uncorrectable errors, RX overflow etc.
- **Bus Master Capability**: Bus Master Capability is essential for DMA functionality. Whether the Bus Master feature is enabled or not can be checked in lspci log.

The lspci output also shows the kernel driver in use at the bottom of its output.
**Figure 8 - lspci sample output log for XDMA**

**Driver Debug**

By default Driver Debug messages are turned off (for better performance). To enable the debug messages, in "include/xdma-core.h", change `#define XDMA_DEBUG 0` to `#define XDMA_DEBUG 1`. This is for the legacy driver. For the latest driver provided in (Xilinx Answer 65444), do the following:

In xdma/libxdma.h file:

- Add `#define __LIBXDMA_DEBUG__`
- Set `XDMA_DEBUG` to `'1'` instead of `'0'`.

After making the change, compile the driver.

The `dmesg` command is used to print out the debug messages from the XDMA driver. Below are the important sections of the debug message log that a user should review when debugging driver issues.

© Copyright 2018 Xilinx
**Driver Load**

- BAR probing
- Channel probing
- Interrupt setup

**DMA Transfer**

- Messages for the entire transfer run
- Prints descriptor dump
- Shows register writes and reads
- Shows interrupt service routine

**BAR Probing**

The driver scans through all of the BARs of the endpoint device and shows which BAR is configured as a DMA BAR. Therefore, in the case of an error when loading the driver, you can check if the DMA configuration BAR is recognised by the driver or not. Figure 9 shows an example of the dmesg log when the BAR configuration is as follows:

- BAR 0: AXI-Lite Interface
- BAR 1: DMA
- BAR 2: DMA Bypass Interface

```plaintext
[ 86.349515] xdma 8800:01.00.0: irq 172 for MSI/MSI-X
[ 85.349547] request_regions():pci_request_regions()
[ 86.349550] map_single_bar():BAR0: 1048576 bytes to be mapped.
[ 86.349608] map_single_bar():BAR0 at 0xdf100000 mapped at 0xffffc9001250000, length=1048576(/1048576)
[ 86.349611] is_config_bar():BAR 0 is not XDMA config BAR, irq_id = 0, cfg_id = 0
[ 86.349612] map_single_bar():BAR1: 65536 bytes to be mapped.
[ 86.349621] map_single_bar():BAR1 at 0xdf200000 mapped at 0xffffc9001200000, length=65536(/65536)
[ 86.349623] is_config_bar():BAR 1 is the XDMA config BAR
[ 86.349624] map_single_bar():BAR2: 1048576 bytes to be mapped.
[ 86.349680] map_single_bar():BAR2 at 0xdf300000 mapped at 0xffffc9001270000, length=1048576(/1048576)
[ 86.349681] map_single_bar():BAR #3 is not present - skipping
[ 86.349682] map_single_bar():BAR #4 is not present - skipping
[ 86.349683] map_single_bar():BAR #5 is not present - skipping
[ 86.349685] set_dma_mask():sizeof(dma_addr_t) == 8
[ 86.349687] set_dma_mask():pci_set_dma_mask()
[ 86.349688] set_dma_mask():Using a 64-bit DMA mask.
[ 86.349700] msi_irq_setup():Using IRQ #49 with 0xffff880036121290
[ 86.349701] msi_irq_setup():Using IRQ #50 with 0xffff8800361212b8
[ 86.349713] msi_irq_setup():Using IRQ #51 with 0xffff8800361212e0
[ 86.349719] msi_irq_setup():Using IRQ #52 with 0xffff880036121308
[ 86.349725] msi_irq_setup():Using IRQ #53 with 0xffff880036121330
```

Figure 9 - BAR Configuration and Interrupt Setup dmesg log

**Interrupt Setup**

After BAR mapping, the interrupts get set up. If all three interrupts (Legacy, MSI, MSI-X) are enabled, the MSI-X interrupts take precedence. Figure 9 shows the IRQ numbers and offset addresses being allocated for the MIS-X interrupt.

**Channel Probing**

After probing the BARs and assigning interrupt numbers, the driver will probe all H2C and C2H DMA channels and create a DMA device for all configured channels. Figure 10 shows the DMA channels being probed by the driver when a single H2C and C2H channel are enabled in the IP. As seen in the log, it will create an engine only if it reads a non-zero engine ID.

© Copyright 2018 Xilinx
DMA Transfer

Figure 11 shows the dmesg log for a DMA transfer.

- **Descriptor Dump**: The driver dumps the Descriptor Field on to the message buffer.
- **Transfer Queue**: Starts DMA Engine
- **Interrupt service routine**: interrupt is serviced
- **DMA status read**: H2C engine status displayed and engine stopped
Case Study

Example 1:

Figure 12 shows an example of a bug where the probing failed when all three BARs (AXI lite, DMA, DMA bypass) are enabled. As shown in the dmesg log, the driver fails to determine BAR 1 as an XDMA config BAR because it returns a config_id of 0 instead of a valid value (0x1fc3). This bug has now been fixed in the IP.
Figure 12 - XDMA Probe Failure

Example 2:

Figure 13 shows a dmesg log when XDMA IP got stuck. It shows the H2C channel in BUSY state. The dmesg log is provided here to illustrate what to check during debugging hang scenarios.

Figure 13 - XDMA IP Busy State

Debug: Register/Ports Reads/Writes

The XDMA IP provides registers to help in debugging DMA issues. These registers can be read using reg_rw command provided with the driver in [Xilinx Answer 65444].

For example, `reg_rw /dev/xdma0_control 0x0000 w` will read register 0x0000

© Copyright 2018 Xilinx
Below are some of the registers that would be useful in debugging:

- Control register (0x0004 for H2C, 0x1004 for C2H)
- Status register (0x0040 for H2C, 0x1040 for C2H)
- Completed descriptor count (0x0048 for H2C, 0x1048 for C2H)
- Interrupt mask
- Interrupt request

To test whether the link is working or not, one could also write and read AXI-Lite Master as shown below:

- `reg_rw /dev/xdma0_user 0x0000 w`: Read
- `reg_rw /dev/xdma0_user 0x0000 w 0x01234567`: Write

**Debug: Status Register**

Table 4 shows the XDMA channel status register. The bits to check in the register are `descriptor_stopped` and `descriptor_completed`. Once transfer completes normally:

- Busy should be 0
- `descriptor_stopped` should be 1
- `descriptor_completed` should be 1

As the status register is cleared on reading once, the above two bits will not be set after the register is read manually.
Table 4 - XDMA Channel Status Register [Ref: PG195]

Debug: Other Registers

In case of a hang or partial completion:

- Check the CDC (completed descriptor count) Register
  - 0x0048 for H2C, 0x1048 for C2H
- Check dmesg to see how many descriptors are generated
- Compare the CDC to the expected descriptor count

If there are no Channel interrupts:

- Check ‘Channel Interrupt Enables Mask’ Register
  - 0x0090 for H2C, 0x1090 for C2H
- Check ‘IRQ Block Channel Interrupt Enable Mask’ Register - Offset: 0x2010
- Read ‘IRQ Block Interrupt Pending’ Register - Offset: 0x204C
  - This shows whether there was interrupt from DMA channels. If it is not set, it means there are no interrupts from the channel source
- Read the ‘IRQ Block Channel Interrupt Request’ Register - Offset: 0x2044
  - If not set, check the ‘IRQ Block Channel Interrupt Enable Mask’ Register

Figure 14 and Figure 15 show the Status, Control, Completed Descriptor Count registers of the H2C and C2H channels read after running the sample application run_test.sh provided in (Xilinx Answer 65444).
While running an application if there is a terminal freeze and XDMA_DEBUG was not enabled in the driver, status registers of the H2C and C2H channel can be read from a parallel terminal. Figure 16 shows a snapshot of H2C0 and C2H0 reads when the channel is actually performing a transfer; the BUSY bit (bit 0) in both channels is set to 1.

Debug: Status Ports

The status ports shown in Table 5 can be enabled from the IP Configuration GUI setting. To understand how the bits in these ports are set, run simulation of the provided example design. Make sure simulation goes through for the selected configuration and check for the corresponding control/status registers that do not get expected values in the system validation. For further debug analysis, check CQ, CC, RQ, RC and AXI interface signals.
Figure 17 shows a snapshot of an ILA capture when H2C data transfer is going on. It shows the h2c_sts busy bit set (bit 0 = 1). Also, note that data being sent out of internal PCIE hard block on the m_axis_rc interface as expected.

**Table 5 - XDMA Channel Status Ports [Ref: PG195]**

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h2c_sts [7:0]</td>
<td>Output</td>
<td>Status bits for each channel. Bit: 6: Control register’s ‘Run’ bit (Table 2.40) 5: IRQ event pending 4: Packet Done event (AXI4-Stream) 3: Descriptor Done event. Pulses for one cycle for each descriptor that is completed, regardless of the Descriptor.Complete field 2: Status register Descriptor_completed bit 1: Status register Descriptor_stopped bit 0: Status register busy bit</td>
</tr>
</tbody>
</table>

Figure 18 shows a snapshot of an ILA capture when C2H data transfer is ongoing. It shows c2h_sts busy bit set (bit 0 = 1). Also, note that data is being input to the internal PCIE hard block on the s_axis_rq interface as expected.
Appendix: Device Driver Source Code Analysis: C2H Transfer

For user reference and illustration purpose, a general analysis of the working mechanism of the device driver source code for C2H has been provided in this section. The source for H2C transfer is similar to the C2H transfer in theory so the details for the H2C transfer are not included. For the latest driver and the corresponding source, refer to (Xilinx Answer 65444).

The driver files for the XDMA IP are:

a. `xdma-core.c`
b. `xdma-core.h`
   a. Header file that defines preprocessor switches, bits of SGDMA control register and bits of the SGDMA descriptor control fields.

A typical C2H flow sequence is as follows:

1. The user program allocates a buffer pointer (based on the size), passes the pointer to the read function with specific device (C2H) and data size.
2. The driver creates descriptors based on transfer length. The following code in the driver does this work.

In `xdma-core.h`, the variables for the descriptor are defined.

```c
struct xdma_desc {
    u32 control;   /* transfer length in bytes */
    u32 bytes;     /* source address (low 32-bit) */
    u32 src_addr_lo; /* source address (high 32-bit) */
    u32 src_addr_hi;
    u32 src_addr_lo; /* destination address (low 32-bit) */
    u32 src_addr_hi;
    u32 dst_addr_lo; /* destination address (high 32-bit) */
    u32 dst_addr_hi;
    u32 next_lo;    /* next desc address (low 32-bit) */
    u32 next_hi;
};
```

© Copyright 2018 Xilinx
In `xdma-core.c`, the following code creates descriptors based on transfer length.

```c
static void xdma_desc_set(struct xdma_desc *desc, dma_addr_t rc_bus_addr, u64 ep_addr, int len, int dir_to_dev)
{
    #if SD_ACCEL
        /* length (in bytes) must be a non-negative multiple of four */
        BUG_ON(len & 3);
    #endif

        /* transfer length */
        desc->bytes = cpu_to_le32(len);
        if (dir_to_dev) {
            /* read from root complex memory (source address) */
            desc->src_addr_lo = cpu_to_le32(PCI_DMA_L(rc_bus_addr));
            desc->src_addr_hi = cpu_to_le32(PCI_DMA_H(rc_bus_addr));
            /* write to end point address (destination address) */
            desc->dst_addr_lo = cpu_to_le32(PCI_DMA_L(ep_addr));
            desc->dst_addr_hi = cpu_to_le32(PCI_DMA_H(ep_addr));
        } else {
            /* read from end point address (source address) */
            desc->src_addr_lo = cpu_to_le32(PCI_DMA_L(ep_addr));
            desc->src_addr_hi = cpu_to_le32(PCI_DMA_H(ep_addr));
            /* write to root complex memory (destination address) */
            desc->dst_addr_lo = cpu_to_le32(PCI_DMA_L(rc_bus_addr));
            desc->dst_addr_hi = cpu_to_le32(PCI_DMA_H(rc_bus_addr));
        }
}
```

The above function is called in `transfer_build()`.

```c
static int transfer_build(struct xdma_transfer *transfer, u64 ep_addr,
                        int dir_to_dev, int non_incr_addr, int force_new_desc,
                        int userspace)
{
    ...........................................
    ...........................................
    xdma_desc_set(transfer->desc_virt + j, cont_addr,
                  ep_addr, cont_len, dir_to_dev);
    ...........................................
    ...........................................
    xdma_desc_set(transfer->desc_virt + j, cont_addr, ep_addr, cont_len,
                  dir_to_dev);
    ...........................................
    ...........................................
    ...........................................
    return j;
}
```

The driver writes the next adjacent descriptor count to 0x5080 if any.

```c
static void xdma_desc_adjacent(struct xdma_desc *desc, int next_adjacent)
{
    int extra_adj = 0;
    /* remember reserved and control bits */
    u32 control = le32_to_cpu(desc->control) & 0x0000f0ffUL;
    u32 max_adj_4k = 0;
```
if (next_adjacent > 0) {
    extra_adj = next_adjacent - 1;
    if (extra_adj > MAX_EXTRA_ADJ) {
        extra_adj = MAX_EXTRA_ADJ;
    }
    max_adj_4k = (0x1000 - ((le32_to_cpu(desc->next_lo)) & 0xFFF))/3);
    if (extra_adj > max_adj_4k) {
        extra_adj = max_adj_4k;
    }
    if (extra_adj < 0) {
        printk("Warning: extra_adj<0, converting it to 0\n");
        extra_adj = 0;
    }
}
/* merge adjacent and control field */
control |= 0xAD4B0000UL | (extra_adj << 8);
/* write control and next_adjacent */
desc->control = cpu_to_le32(control);
}

The above function is called in the xdma_transfer function.

static struct xdma_transfer *transfer_create(struct xdma_dev *lro,
const char *start, size_t cnt, u64 ep_addr, int dir_to_dev, int non_incr_addr, int force_new_desc, int userspace) {
………………………………………………
………………………………………………
/* fill in adjacent numbers */
for (i = 0; i < transfer->desc_num; i++) {
    xdma_desc_adjacent(transfer->desc_virt + i,
    transfer->desc_num - i - 1);
}
/* initialize wait queue */
init_waitqueue_head(&transfer->wq);
return transfer;
}

The driver starts a C2H transfer by writing to the C2H engine control register address 0x1004.

In xdma-core.h, the engine control register is defined as follows:

struct engine_regs {
    u32 identifier;
    u32 control; /* This the C2H channel control register (0x0004)*/
    u32 control_w1s;
    u32 control_w1c;
    u32 reserved_1[12];/* padding */
    u32 status;
    u32 status_rc;
    u32 completed_desc_count;
    u32 alignments;
    u32 reserved_2[14];/* padding */
    u32 poll_mode_wb_lo;
    u32 poll_mode_wb_hi;
    u32 interrupt_enable_mask;
    u32 interrupt_enable_mask_w1s;
    u32 interrupt_enable_mask_w1c;
    u32 reserved_3[9]; /* padding */
    u32 perf_ctrl;
© Copyright 2018 Xilinx
The following code in `xdma-core.c` writes to the control register of the C2H engine.

```c
static void engine_start_mode_config(struct xdma_engine *engine) {
    /* write control register of SG DMA engine */
    u32 w = (u32)XDMA_CTRL_RUN_STOP;
    w |= (u32)XDMA_CTRL_IE_READ_ERROR;
    w |= (u32)XDMA_CTRL_IE_DESC_ERROR;
    w |= (u32)XDMA_CTRL_IE_DESC_ALIGN_MISMATCH;
    w |= (u32)XDMA_CTRL_IE_MAGIC_STOPPED;
    if (poll_mode) {
        w |= (u32)XDMA_CTRL_POLL_MODE_WB;
    } else {
        w |= (u32)XDMA_CTRL_IE_DESC_STOPPED;
        w |= (u32)XDMA_CTRL_IE_DESC_COMPLETED;
        /* enable IE_IDLE_STOP only for AXI ST C2H and for perf test */
        if (engine->streaming && !engine->dir_to_dev) {
            w |= (u32)XDMA_CTRL_IE_IDLE_STOPPED;
        }
        if (engine->xdma_perf) {
            w |= (u32)XDMA_CTRL_IE_IDLE_STOPPED;
        }
    }
}
```

The engine is started by the following function:

```c
static struct xdma_transfer *engine_start(struct xdma_engine *engine) {
    /* iterate over C2H (PCIe write) */
    for (channel = 0; channel < XDMA_CHANNEL_NUM_MAX; channel++) {
        engine = lro->engine[channel][1];
        /* engine present and its interrupt fired? */
        if (engine && (engine->irq_bitmask & channel)) {
            dbg_tfr("schedule_work(engine=%p)\n", engine);
            schedule_work(&engine->work);
        }
    }
}
```

After the hardware sends an interrupt to the host, the driver acts again. It reads 'IRQ Block Channel Interrupt Request' 0x2044 to see which channel sent the interrupt.

```c
static irqreturn_t xdma_isr(int irq, void *dev_id) {
    /* iterate over C2H (PCIe write) */
    for (channel = 0; channel < XDMA_CHANNEL_NUM_MAX; channel++) {
        engine = lro->engine[channel][1];
        /* engine present and its interrupt fired? */
        if (engine && (engine->irq_bitmask & ch irq)) {
            dbg_tfr("schedule_work(engine=%p)/", engine);
            schedule_work(&engine->work);
        }
    }
}
```
Disable channel interrupt by writing to 0x2018 as needed.

```c
/* channel_interrupts_disable -- Disable interrupts we are not interested in */
static void channel_interrupts_disable(struct xdma_dev *lro, u32 mask)
{
    struct interrupt_regs *reg = (struct interrupt_regs *)
        (lro->bar[lro->config_bar_idx] + XDMA_OFS_INT_CTRL);
    write_register(mask, &reg->channel_int_enable_w1c);
}
```

The Driver reads the corresponding 'Status Register' 0x1044 which will also clear the status register.

```c
static int engine_service(struct xdma_engine *engine, int desc_writeback)
{
    ............................................................
    ............................................................
    /* Service the engine */
    if (!engine->running) {
        dbg_tfr("Engine was not running!!! Clearing status\n");
        if (desc_writeback == 0)
            engine_status_read(engine, 1);
        return 0;
    }
    ............................................................
    ............................................................
    return rc;
}
```

The `engine_status_read` function is defined as follows.

```c
static u32 engine_status_read(struct xdma_engine *engine, int clear)
{
    u32 value;
    BUG_ON(!engine);
    engine_reg_dump(engine);
    /* read status register */
    dbg_tfr("Status of SG DMA %s engine:\n", engine->name);
    dbg_tfr("ioread32(0x%p):\n", &engine->regs->status);
    if (clear) {
        value = engine->status =
            read_register(&engine->regs->status_rc);
    } else {
        value = engine->status = read_register(&engine->regs->status);
    }
    dbg_tfr("status = 0x%08x: %s%s%s%s%s%s%s\n", (u32)engine->status,
        (value & XDMA_STAT_BUSY) ? "BUSY " : "IDLE ",
        (value & XDMA_STAT_DESC_STOPPED) ? "DESC_STOPPED " : "",
        (value & XDMA_STAT_DESC_COMPLETED) ? "DESC_COMPLETED " : "",
        (value & XDMA_STAT_ALIGN_MISMATCH) ? "ALIGN_MISMATCH " : "",
        (value & XDMA_STAT_MAGIC_STOPPED) ? "MAGIC_STOPPED " : "",
        (value & XDMA_STAT_FETCH_STOPPED) ? "FETCH_STOPPED " : "");
}
```
The code reads the Read Channel 'Completed descriptor count' register at 0x1048.

```c
if (desc_count == 0)
    desc_count = read_register(&engine->regs->completed_desc_count);
```

The completed descriptor count is defined in the `xdma.c` file as shown below.

```c
struct engine_regs {
    u32 identifier;
    u32 control;
    u32 control_w1s;
    u32 control_w1c;
    u32 reserved_1[12]; /* padding */
    u32 status;
    u32 status_rc;
    u32 completed_desc_count;
    u32 alignments;
    u32 reserved_2[14]; /* padding */
    u32 poll_mode_wb_lo;
    u32 poll_mode_wb_hi;
    u32 interrupt_enable_mask;
    u32 interrupt_enable_mask_w1s;
    u32 interrupt_enable_mask_w1c;
    u32 reserved_3[9]; /* padding */
    u32 perf_ctrl;
    u32 perf_cyc_lo;
    u32 perf_cyc_hi;
    u32 perf_dat_lo;
    u32 perf_dat_hi;
    u32 perf_pnd_lo;
    u32 perf_pnd_hi;
} __packed;
```

The code below shows a write to the 'Channel Control Register' at 0x1004 to stop DMA run.

```c
/**
 * xdma_engine_stop() - stop an SG DMA engine
 */
static void xdma_engine_stop(struct xdma_engine *engine)
{
    u32 w;
    BUG_ON(!engine);
    dbg_tfr("xdma_engine_stop(engine=%p)\n", engine);
    w = 0;
    w |= (u32)XDMA_CTRL_IE_DESC_ALIGN_MISMATCH;
```
w |= (u32)XDMA_CTRL_IE_MAGIC_STOPPED;
w |= (u32)XDMA_CTRL_IE_READ_ERROR;
w |= (u32)XDMA_CTRL_IE_DESC_ERROR;

if (poll_mode) {
    w |= (u32) XDMA_CTRL_POLL_MODE_WB;
} else {
    w |= (u32)XDMA_CTRL_IE_DESC_STOPPED;
    w |= (u32)XDMA_CTRL_IE_DESC_COMPLETED;

    /* Disable IDLE STOPPED for MM */
    if ((engine->streaming && (engine->dir_to_dev == 0)) ||
        (engine->xdma_perf))
        w |= (u32)XDMA_CTRL_IE_IDLE_STOPPED;
}

dbg_tfr("Stopping SG DMA %s engine; writing 0x%08x to 0x%p.\n", engine->name, w, (u32 *)&engine->regs->control);
write_register(w, &engine->regs->control);
/* dummy read of status register to flush all previous writes */
dbg_tfr("xdma_engine_stop(%s) done\n", engine->name);

Write to `Block channel interrupt Enable Mask` 0x2014 to enable interrupt for next transfer.

/ * channel_interrups_enable -- Enable interrupts we are interested in */
static void channel_interrups_enable(struct xdma_dev *lro, u32 mask)
{
    struct interrupt_regs *reg = (struct interrupt_regs *)
        (lro->bar[lro->config_bar_idx] + XDMA_OFS_INT_CTRL);

    write_register(mask, &reg->channel_int_enable_w1s);

Return control to the application program with the transfer size.

static void __exit xdma_exit(void)
{
    dbg_init(DRV_NAME" exit(\n");
    /* unregister this driver from the PCI bus driver */
    pci_unregister_driver(&pci_driver);
    if (g_xdma_class)
        class_destroy(g_xdma_class);
}

module_init(xdma_init);
module_exit(xdma_exit);

Reference

- (PG195): DMA/Bridge Subsystem for PCI Express v4.1
- (Xilinx Answer 65444): Xilinx PCI Express Drivers and Software Guide