# DMA transfers between PL and PS on MPSoC from user space without custom kernel driver

H. Sandberg (CERN SY-BI-XEI), 2022-05-03
https://indico.cern.ch/event/1139381/

# Motivation

Direct memory access (DMA) speeds up data transfer between the programmable logic (PL) and the processing system (PS) running Linux on the MPSoC.
No “generic” DMA driver seems to exist which can be used from an application in user space.

**Custom kernel driver is usually used!**

As an electronics hardware engineer I have limited/no experience with writing Linux kernel drivers. The next person working on this project will probably have a similar background as me. Debugging and writing a stable kernel driver seems tricky also...

**Can we do DMA transfers from/to user space with existing kernel drivers?**

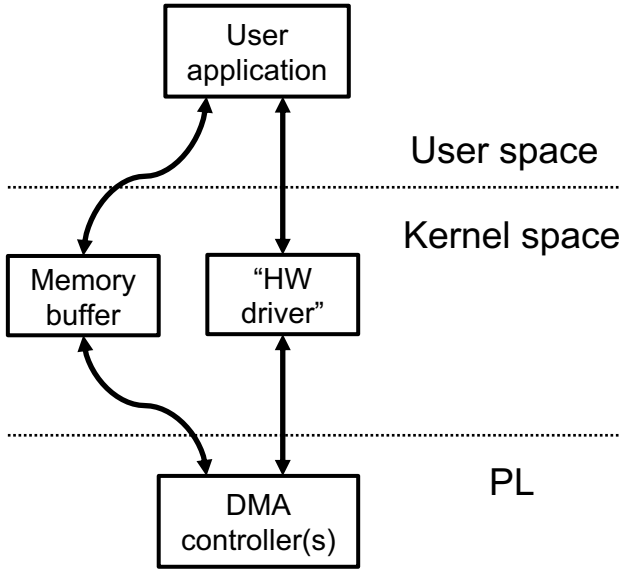
# DMA in a nutshell

There are two parts to using DMA in the MPSoC with Linux:

1. Configuration of the DMA controller in the PL
2. Allocation of continuous memory buffer in kernel space

A custom kernel driver can take care of both of these and present a custom interface to user space where data is passed through

A user space driver/application does not have access to the physical memory and needs some way to access the DMA controller hardware



# Kernel driver examples/tutorials

## Xilinx example ”Linux DMA from User Space”

- Actually requires a custom ”DMA proxy” kernel driver so not really user space
- https://www.xilinx.com/video/soc/linux-dma-from-user-space.html
- https://github.com/Xilinx-Wiki-Projects/software-prototypes/tree/master/linux-user-space-dma
- https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842418/Linux+DMA+From+User+Space

## Bare metal (i.e. not Linux)

- https://www.fpgadeveloper.com/2014/08/using-the-axi-dma-in-vivado.html/

# User space driver examples/tutorials

## DMA controller access with /dev/mem:

- https://lauri.võsandi.com/hdl/zynq/xilinx-dma.html
- https://support.xilinx.com/s/article/1223569
- https://www.hackster.io/whitney-knitter/introduction-to-using-axi-dma-in-embedded-linux-5264ec
- **Recommendation seems to be to avoid devmem for operational systems**

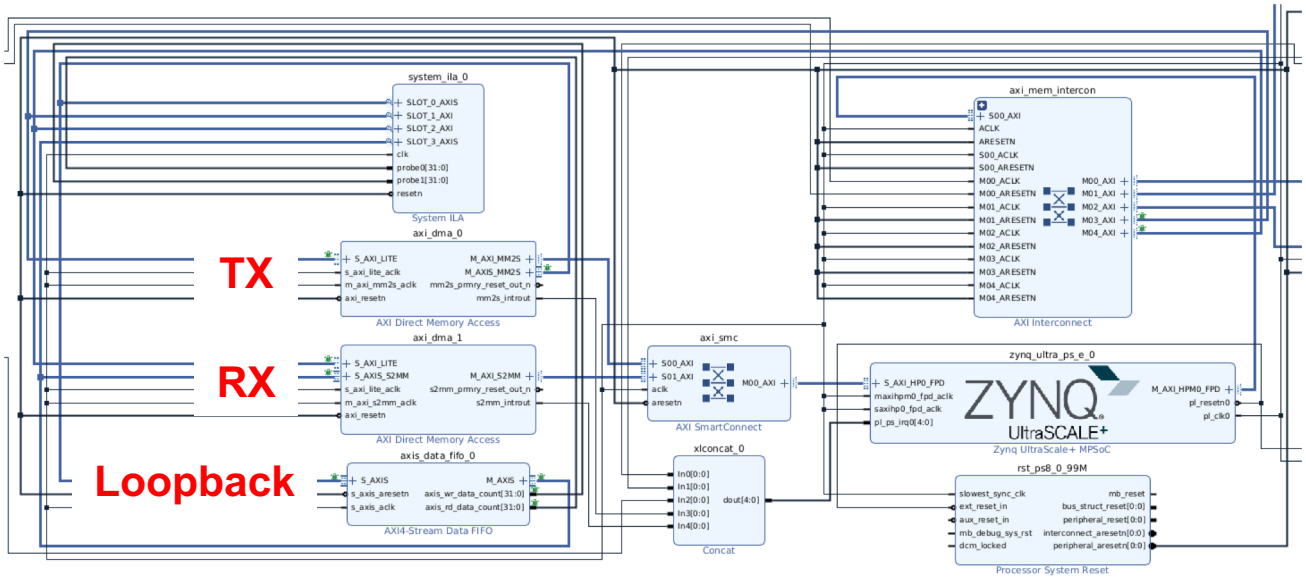
## DMA controller access with UIO:

- https://github.com/Kirill888/parallella-fpga-dummy-io/tree/master/sample_dma
- https://www.bastibl.net/futuresdr-2/
  - I used this example as a base but used Python instead of Rust for the user space application

## Different ways for memory buffer:

- Some use reserved memory region in devicetree
- u-dma-buf kernel driver seems like a better way (see later slide)

# MPSoC block diagram




- axi_dma_0: TX, axi_dma_1: RX - with separate interrupts
- AXI_LITE used to configure DMA controllers
- Loopback via AXI4-stream FIFO

# User space accessible memory buffer

**u-dma-buf:** https://github.com/ikwzm/udmabuf

- Several ways to instantiate (see github readme)
- I'm using the devicetree
  - Actually devicetree overlay loaded together with the PL gateware after Linux boot
- Creates a buffer in kernel space with continuous memory and exposes the physical address to user space via a file

```
// One 64 MB u-dma-buf buffer (User space mappable DMA Buffer)
udmabuf@0 {
  compatible = "ikwzm,u-dma-buf";
  device-name = "udmabuf0";
  minor-number = <0>;
  size = <0x4000000>;
  sync-mode = <1>;
  sync-always;
};
```

```
root@bipxl-petalinux:/# cat /sys/class/u-dma-buf/udmabuf0/phys_addr
0x000000006e100000
```

# User space accessible memory buffer

- Memory buffer can be accessed from Python using numpy memmap
- Writing data to the buffer is non-volatile, i.e. it stays there the next time you read it back

```
>>> import numpy as np
>>> a = np.memmap('/dev/udmabuf0', dtype=np.uint8, mode='r+', shape=(67108864))
>>> a
memmap([0, 0, 0, ..., 0, 0, 0], dtype=uint8)
>>> a[0] = 1
>>> a
memmap([1, 0, 0, ..., 0, 0, 0], dtype=uint8)
```

```
root@bipxl-petalinux:/# dd if=/dev/udmabuf0 of=udmabuf0.bin bs=8388608
8+0 records in
8+0 records out
67108864 bytes (67 MB, 64 MiB) copied, 1.12539 s, 59.6 MB/s
root@bipxl-petalinux:~/udmabuf# od -t x1 udmabuf0.bin
0000000 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
400000000
```

https://numpy.org/doc/stable/reference/generated/numpy.memmap.html

# User space configuration of DMA controller

Change driver in devicetree to "generic-uio”

AXI DMA controller datasheet

- Table 2-6 shows the register map we will use to configure the controller
- https://docs.xilinx.com/v/u/en-US/pg021_axi_dma

Config in a nutshell (TX shown here):

- Clear IRQs (MM2S_DMASR)
- Set config (MM2S_DMACR)
- Set address (MM2S_SA)
- Set length (MM2S_LENGTH)

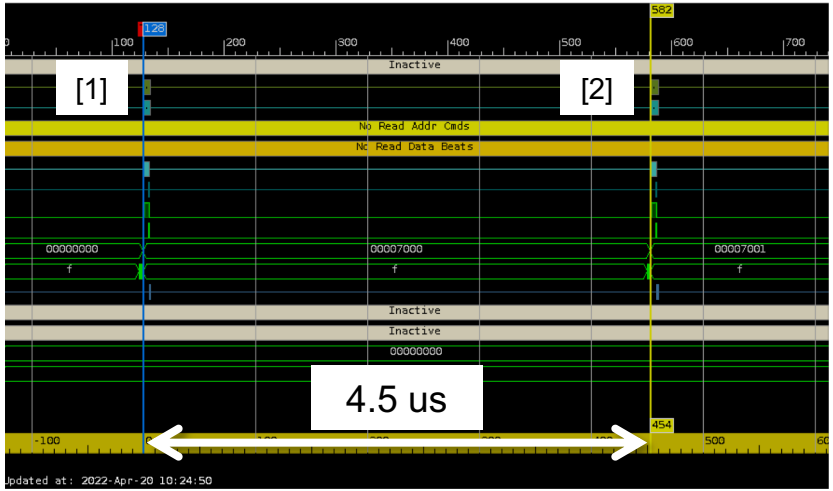
Transfers starts automatically after length is set

```
axi_dma_0: dma@a8020000 {
  #dma-cells = <1>;
  clock-names = "s_axi_lite_aclk", "m_axi_mm2s_aclk";
  clocks = <&zynqmp_clk 71>, <&zynqmp_clk 71>;
  // compatible = "xlnx,axi-dma-7.1", "xlnx,axi-dma-1.00.a";
  compatible = "generic-uio";
  interrupt-names = "mm2s_introut";
  interrupt-parent = <&gic>;
  interrupts = <0 92 4>;
  reg = <0x0 0xa8020000 0x0 0x10000>;
  xlnx,addrwidth = <0x40>;
  xlnx,sg-length-width = <0x1a>;
  label = "axi_dma_0_tx";
  linux,uio-name = "axi_dma_0_tx";
  dma-channel@a8020000 {
    compatible = "xlnx,axi-dma-mm2s-channel";
    dma-channels = <0x1>;
    interrupts = <0 92 4>;
    xlnx,datawidth = <0x20>;
    xlnx,device-id = <0x0>;
  };
};
```

# User space configuration of DMA controller

## How long does it take to configure?

- Each UIO write command takes some time to reach the AXI-lite port of the DMA controller
- Waveform shows 4.5 us from one command to the next




```
tx_uio.write_uint32(DMA_REG_MM2S_DMASR, 0x7000) # [1]
tx_uio.write_uint32(DMA_REG_MM2S_DMACR, 0x7001) # [2]
tx_uio.write_uint32(DMA_REG_MM2S_SA, (tx_udmabuf.phys_addr & 0xFFFF_FFFF))
tx_uio.write_uint32(DMA_REG_MM2S_SA_MSB, ((tx_udmabuf.phys_addr >> 32) & 0xFFFF_FFFF))
tx_uio.write_uint32(DMA_REG_MM2S_LENGTH, num_bytes_to_transfer)
```

# Preliminary performance tests - loopback

Test setup with FIFO loopback between TX and RX DMA:

1. Clear TX and RX buffers to zeros
2. Fill TX buffer with 32 MB of random uint64 numbers
3. Start timer
4. Configure TX DMA and start it
5. Configure RX DMA and start it
6. Wait for RX DMA interrupt
7. Stop timer
8. Compare TX and RX buffers

**Total time:**
**~85 ms (32-bit AXI4-stream)**
**~55 ms (64-bit AXI4-stream)**
**~49 ms (128-bit AXI4-stream)**

# Preliminary performance tests - RX

PL -> PL DDR -> FIFO -> DMA -> PS DDR -> Python

- Packet generator in PL writes to PL DDR
- State machine reads from PL DDR and pushes to a FIFO as soon as there is any space
- Empty all available data at the moment from FIFO with RX DMA
- Data ends up in Python numpy array
- No processing of the data (potential future bottleneck)
- Repeat

**~80 Mbyte/s**

Current bottleneck is size of FIFO and AXI4-stream construction

To-be-continued...

# Questions and comments?

To-be explored and tested:

- Scatter-gather
- Caching and coherency

For info:

- BIPXL Zynq MPSoC Readout EDMS project: https://edms.cern.ch/project/CERN-0000223027