LINUX general purpose PCIe driver for MTCA

4th MTCA Workshop for Industry and Research

Davit Kalantaryan
4th MTCA Workshop
DESY, 11.12.2015

Content

Introduction (necessity of general driver)

MTCA is going to be the standard for some decades. One of the important and time consuming task is the software development for MTCA. It is obvious that user space software development is preferable in comparison with kernel driver development, because of

- **Debugging**
- **Possibility to use more high level programming languages, that improve productivity and decrease possibility for errors**
- **Easy maintenance and adaptation in the case of LINUX kernel changes**

Therefore it is worthwhile to engage couple of years in the development of kernel space general purpose driver based on MTCA standards. Ultimately only user space software will be required to adopt new MTCA devices. The design and the development of the general driver started in 2013 with the objective of finally creating a driver that is able to handle as many MTCA devices as possible. The following rule is always kept: if a generalization of any functionality leads to penalty in performance or increase in memory usage or too complicated code is abandoned. In the case a device is not possible to be handled by this driver due to device very specific functionality, driver stacking can be used. The driver can be parent driver for specific device driver.

Hardware developer also can use the driver to make tests during hardware development.

Introduction (when a driver is required)

Devices are controlled using their IO registers. Usually device behavior can be changed by writing some value to the devoted register. If only read and write operations are sufficient to control the device, no driver is needed. In this case one can simply use **`/dev/mem´** entry for mapping IO device memory and registers to user process address space. In this way the whole task is completed.

Undeniably, if there is a possibility to make a device handling software without the kernel space driver development (avoiding a penalty in performance and CPU usage), user space programming should be preferred.

Unfortunately, in some cases the sole user space software is not sufficient for handling the device, and even worse, for each device requirements for kernel driver can vary (as a consequence a great deal of development and independent driver for each device are needed. Next slide represents functionalities when kernel space driver is required).

MTCA is being effectively used in DESY. A driver that is able to handle a lot of MTCA devices is being developed. The idea is that MTCA devices have to satisfy some standards, and based on these hardware generalities (driven by standard) a driver that is able to handle many AMC4 compliant boards can be created.

Functionalities that force driver usage

- Atomic access to several registers (read/read,read/write,…)
- Error handling
- Hot plug handling
- Device interrupts handling
- Initiating DMA (if device is capable to make DMA)

So the statement is following: If there is a driver, that is able to handle these issues in an effective way for many MTCA devices, then most of (these) devices can be handled by this driver and adding new device to MTCA will cost only user space software development.

- Currently first 3 points are implemented in general purpose driver.
- Already debug version of driver with common IRQ handling routine exists. Some devices in Zeuthen use of the IRQ functionality of the driver.
- DMA: In design stage

Requirements to the driver

> Due to the generalization of any functionality no performance penalty should occur.

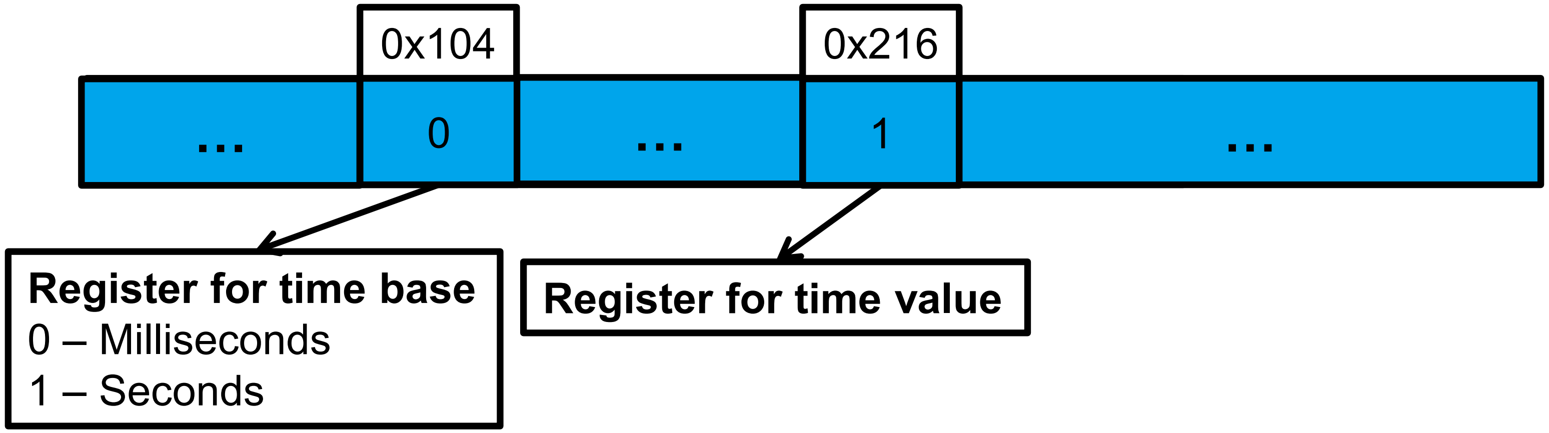
> There should not be any considerable increase in memory usage due to the general functionality. If there is a functionality that cannot be implemented without performance lowering or memory usage increase, it will not be added to the driver.

> The code should not be long and complicated. Otherwise no benefit will be achieved because of the difficult maintenance.

> Driver must export all necessary interfaces for other top level drivers (driver stacking). Then in the case of specific device, that is not possible to handle with this driver, one can easily create driver for this specific device on top of this driver with less effort.

Example of usage of atomic access to several registers (read/read,write/write,read/write,…)

Assume there is a timer device and setting time should be done by first setting time base and then value. If these 2 set value operations are not done in atomic manner a following problem can occur: after setting time base by program A, program B changes time base and then again program A sets value with wrong time base



Prog. A) Setting time to 1 second
1. **Set(0x104,1);** - setting time base to s
2. **Set(0x216,1);** - setting time

Prog. B) Setting time to 900 milliseconds
1. **Set(0x104,0);** - setting time base to ms
2. **Set(0x216,900);**- setting time

Possible sequence of actions performed by two concurrent programs
1. B) **Set(0x104,0);** // In this case finally time will be 900s instead of being ~1s
2. A) **Set(0x104,1);**
3. A) **Set(0x216,1);** // this may lead to serious problems in system
4. B) **Set(0x216,900);**

Vectorised device access in atomic manner

As a solution to the problems described in previous slide, ioctl call was implemented to make several device accesses at once. The types of these device accesses are

a) Read one or more registers
b) Write one or more registers
c) Set any amount of bits of any amount of registers
d) Swap any amount of bits of any amount of registers

Structure to make this ioctl call is following

```
ioctl(fd,PCIEDEV_VECTOR_RW,aRWData);
```

```
typedef struct device_vector_rw
{
    u_int64_t      number_of_rw;      /* number of device accesses                  */
    pointer_type   device_ioc_rw_ptr; /* pointer to the data for all device accesses */
} device_vector_rw;
```

In the first field (number_of_rw) number of atomic accesses to device provided. The second field (device_ioc_rw_ptr) is pointer to the device access data of this type

```
typedef struct device_ioc_rw
{
    u_int16_t      register_size_mode; /* RW_D08, RW_D16, RW_D32  , if<0, then default is used          */
    u_int16_t      rw_access_mode;     /* (read,write,set-bits,swap-bits)                                */
    u_int32_t      barx_rw;            /* BARx (0, 1, 2, 3, 4, 5)                                        */
    u_int32_t      offset_rw;          /* offset in address                                              */
    u_int32_t      count_rw;           /* number of register to handle                                   */
    pointer_type   dataPtr;            /* pointer to the buffer for writing to device or to store data from device */
    pointer_type   maskPtr;            /* pointer to the buffer for mask for bitwise operations          */
}device_ioc_rw;
```

Several IO operations with intermediate calculations

When several register accesses have to be atomic and they are dependent on each other, it presents an even more difficult case.

In the case of several dependent IO operations each register access depends on the result from the previous accesses. After each operation, some calculations should be performed for preparing next IO operation.

Another example for timer device → if time base is ms then value is set to 1000 and set to 1 if the base is second.

Prog. A) Setting time to 1 second (WCTB)

```
1.  timeBase = Get(0x104);        - set time base
2.  setValue = timeBase==0 ? 1000 : 1;
3.  Set(0x216,setValue);          - setting time
```

Prog. B) Setting time to 1 second (CTB)

```
1.  Set(0x104,1);     - setting time base to s
2.  Set(0x216,1);     - setting time
```

Possible sequence of actions performed by two concurrent programs

1. Initial time base = 0;
2. A) **timeBase = Get(0x104);** **(timeBase=0) => 1000 will be set**
3. B) **Set(0x216,1000);**
4. B) **Set(0x104,1);**
5. A) **(timeBase=0) => 1000 will be set** **// Intermediate calculation**
6. A) **Set(0x216,1000);** // In this case final time will be 1000s instead of being 1s

Many operations in sequence with intermediate calculations

For implementing sequential accesses with intermediate calculations, two ioctl calls are implemented for both locking and unlocking device. Between locking and unlocking the program can make any amount of system calls to driver for accessing device. Driver checks if the TID of thread corresponds to the TID of locker thread then all operations take place without locking. Meanwhile, all the other programs wait. **This locking has configurable timeout. Whenever this timeout is reached and lock is not released by the program, the kernel driver will release the semaphore lock for this program and send interrupt signal.**

1. ioctl(fd,PCIEDEV_LOCK_DEVICE);
2. read(), some calculations, write() again some calc. set-bits, ….
3. ioctl(fd,PCIEDEV_UNLOCK_DEVICE);

This approach is a little bit similar to **flock** standard system call http://linux.die.net/man/2/flock and may be in the future instead of using ioctl for locking device this system call will be used.

Whenever it is possible vectorised device accesses should be preferred to the scheme mentioned above. For this scheme the intermediate calculations are done in user space and several context switches between kernel space and user space take place which can slow down the performance.

Bitwise operations

For bitwise operations the driver provides 2 system calls

> ioctl(fd,PCIEDEV_SET_BITS,data): For setting any amount of bits from any amount of registers

> ioctl(fd,PCIEDEV_SWAP_BITS,data): For swapping any amount of bits from any amount of registers.

Data to be provided with these ioctl calls should be the pointer to the following structure

```
typedef struct device_ioc_rw
{
    u_int16_t     register_size_mode; /* RW_D08, RW_D16, RW_D32                                 */
    u_int16_t     rw_access_mode;     /* (read,write,set-bits,swap-bits)                        */
    u_int32_t     barx_rw;            /* BARx (0, 1, 2, 3, 4, 5)                                */
    u_int32_t     offset_rw;          /* offset in address                                      */
    u_int32_t     count_rw;           /* number of register to handle                           */
    pointer_type  dataPtr;            /* pointer to the buffer for writing to device or to store data from device */
    pointer_type  maskPtr;            /* pointer to the buffer for mask for bitwise operations  */
}device_ioc_rw;
```

For these ioctl calls field **rw_acces_mode** is ignored, because mode is set by kernel driver to set-bits or swap-bits correspondingly. If register size is negative, then driver will use default register size for corresponding device. For swap-bits operations **dataPtr** field is not used; can have any value

read/write, pread/pwrite

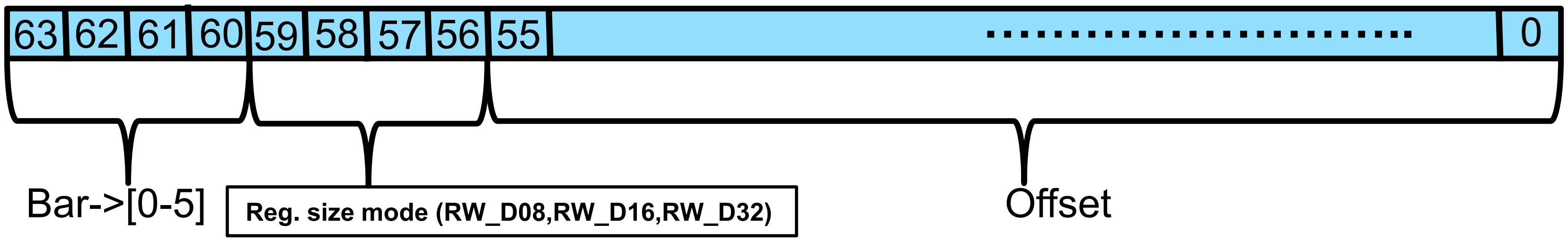
During read, write, pread or pwrite system calls the kernel space driver should know the pci bar (0-5) and the offset. In the case of read/write system calls user space application provides this information in the structure presented in the next slide. In the case of pread/pwrite this information is provided in the fourth argument (offset) of these system calls. C syntaxes for the functions implementing these system calls are following

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

Register size is also provided in the offset field, but register size is known by the driver. So if negative register size is provided, then correct register size will be selected by the driver.
So the bits for the bar, the offset and the register size mode are the following

Bits [0-55]	keep offset	???
Bits [56-59]	keep register size	??????
Bits [60-63]	keep mode	???

Instead of using these number better to use following macroses
PRW_REG_SIZE_MASK, PRW_BAR_MASK, PRW_OFFSET_MASK, PRW_REG_SIZE_SHIFT, PRW_BAR_SHIFT



Structure for read/write system calls

In the case of read/write system calls all necessary information is provided by the structure below.

```
typedef struct device_rw
{
    u_int32_t      offset_rw;       /* offset in address                                          */
    u_int32_t      data_rw;         /* data to set or return data in the case of 1 register access */
    union
    {
        u_int32_t      mode_rw;     /* RW_D08, RW_D16, RW_D32.  if<0, then default                */
        u_int32_t      register_size; /* RW_D08, RW_D16, RW_D32. if<0, then default               */
    };
    u_int32_t      barx_rw;         /* BARx (0, 1, 2, 3, 4, 5)                                    */
    union
    {
        struct
        {
            u_int32_t      size_rw;   // !!! transfer size should not be providefd by this field.
                                      // This field is there for backward compatibility.
                                      //
                                      // Transfer size should be provided with read/write 3-rd
                                      // argument (count)
                                      // read(int fd, void *buf, size_t count);
                                      // ssize_t write(int fd, const void *buf, size_t count);
            u_int32_t     rsrvd_rw;   // Not used
        };
        pointer_type  dataPtr;      // pointer to the buffer for writing to device or to store data from device
                                    // when number of registers more than one
    };
}device_rw;
```

Interrupt handling

For handling interrupt from any device, following steps should be confirmed

> The first step is to understand if interrupt comes from our device (IRQ_HANDLED) or from another device (IRQ_NONE) that is sharing same interrupt line (because nowedays interrupt lines are mainly shared)

> Read all necessary information, for this interrupt

> Inform user space applications about interrupt

> Acknowledge device

Handling device interrupts

Following steps should be done

1. **first is to understand if interrupt comes from our device or from another device, that is sharing same interrupt line (because now days interrupt lines are mainly shared)**
2. **Read all necessary information, for this interrupt**
2. **Acknowledge device**
3. **Inform user space applications about interrupt**

All these steps are implemented in such a way, that for most devices, the implementation works.

1. One register should be provided to the driver, that shows if device has made interrupt or not.
2. For reading interrupt information and acknowledging device some registers and corresponding operations types (read/write/bitwise) with corresponding values (in the case of write) are provided
3. Two methods are implemented to inform the user space application about a device interrupt: a) sending a signal to all interested user space applications (from deferred bottom half interrupt handler), b) waking up all interested applications

```
struct device_irq_handling aIrqHandleInfo;
…. //Filling info
::ioctl(fd, GEN_REQUEST_IRQ_FOR_DEV,&aIrqHandleInfo);
```

Timing

Performance tests for the following generalized functionalities have been done

1. Register accesses (read/write/ioc-rw)
2. IRQ handling

Case of a driver per device

- read(100regs)-> ~240 ms
- write(100regs)-> ~14 ms
- Irq_handling -> ~700 ns

Case of a driver for all device

- read(100regs)-> ~240 ms
- write(100regs)-> ~14 ms
- Irq_handling -> ~800 ns

Memory usage

Case of a driver per device

llrfadc.ko	->	~30 kB
llrfdamc.ko	->	~30 kB
llrfutc.ko	->	~30 kB
sis8300.ko	->	~38 kB
x1timer.ko	->	~49 kB

Sum.: 177 kB

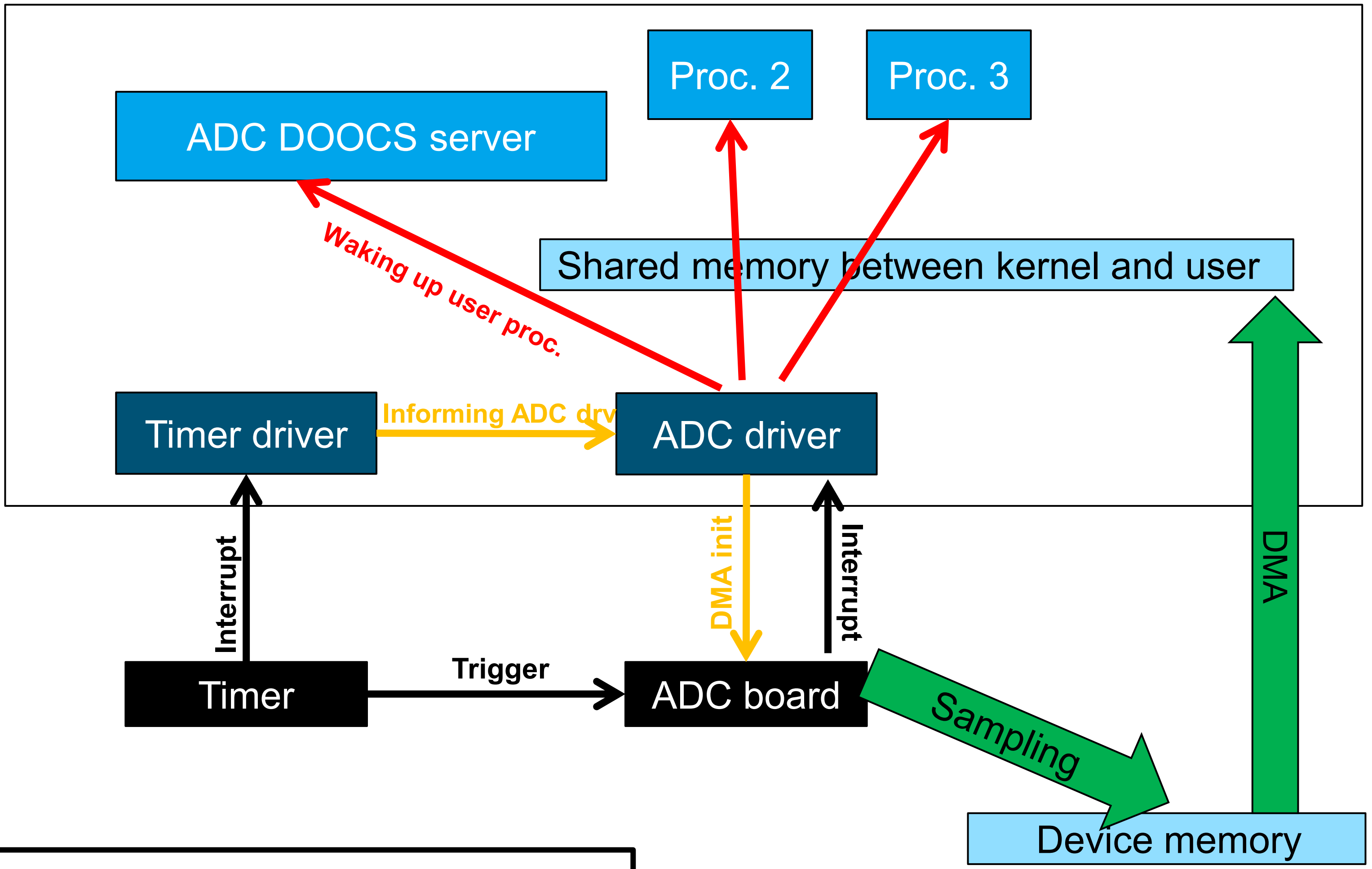
Case of a driver for all device

mtcagen.ko	->	~100 kB

Sum.: 100 kB

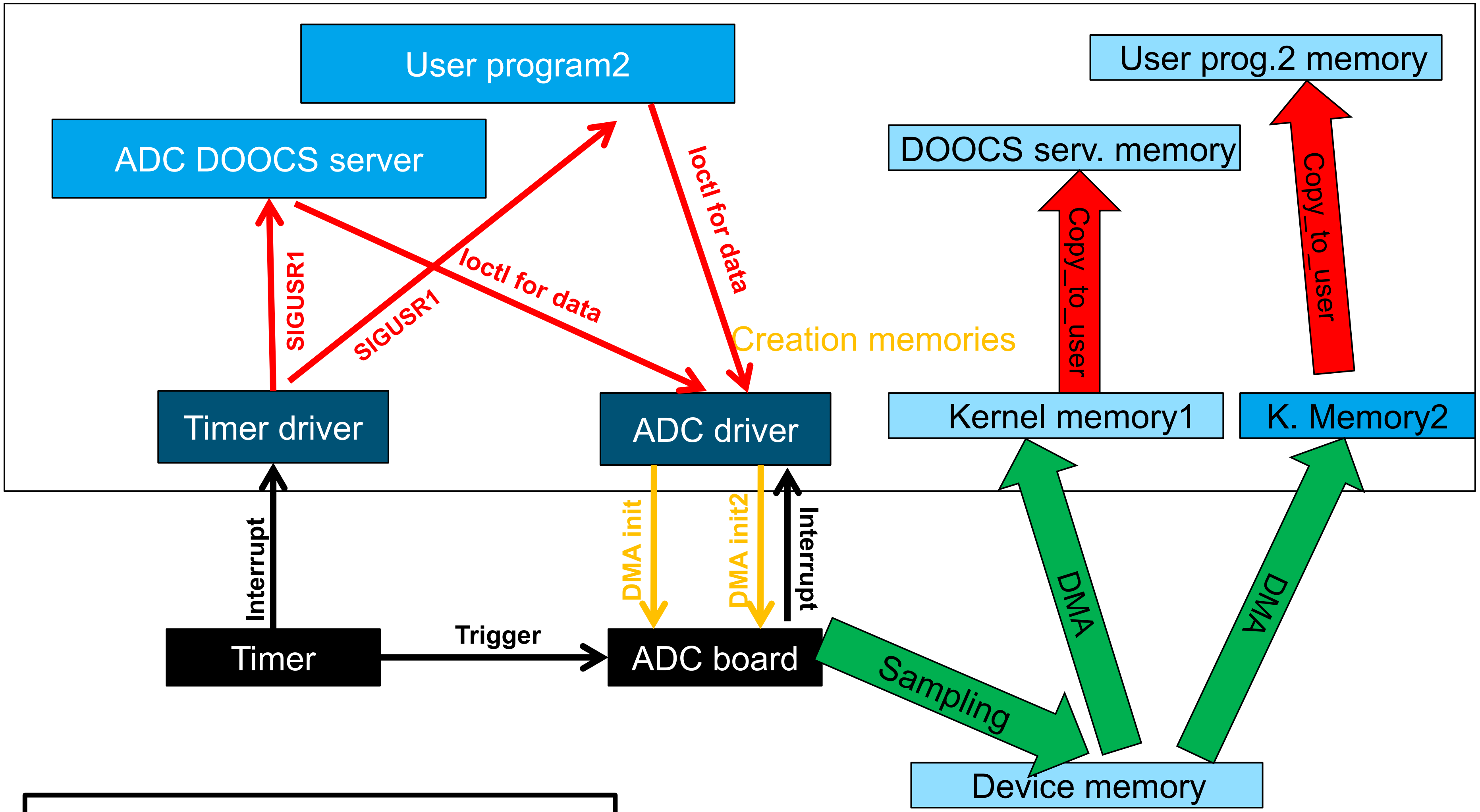
Another contribution to the memory usage comes from dynamically allocated memory by a driver. Dynamic allocation of memory by the driver is minimized and it is really neglectable.

Data and notification flows for coherent mapped buffer case



```
wake_up (&(dev->waitDMA));
```

Error prone cases for streaming buffer usage



```
wake_up (&(dev->waitDMA));
```

Comparison of steps for streaming and coherent mapped buffer cases

Black actions are not done by CPU.
Yellows: done by CPU
Red: done by CPU and CPU usage is high (context switches and huge data copy)

a) Timer module triggers the ADC board to start sampling
b) After some configurable timeout timer module generates interrupt
c) Kernel space timer driver sends SIGUSR1 signal to the DOOCS server responsible for ADCs
d) DOOCS server makes ioctl call to ADC kernel driver for data and waits
e) ADC driver requests DMAable memory from the system (get_free_pages(GFP_DMA,…))
f) ADC driver maps requested buffer for DMA (disables Read cache)
g) ADC driver initiates device for making DMA the sampled data and waits.
h) After DMA is done ADC board device interrupts the CPU
i) ADC driver copies data from kernel buffer to user buffer (copy_to_user)!!!
j) Finally ADC driver unmaps the buffer and returns it to system

Black actions are not done by CPU.
Yellows: done by CPU
Red: done by CPU and CPU usage is high (context switches and huge data copy)

a) Timer module triggers the ADC board to start sampling
b) After some configurable timeout the timer module generates interrupt
c) Timer driver informs ADC driver, that sampling is done
d) ADC driver prepares DMA to the already created system memory. This memory is shared with user space applications
e) After DMA is done device interrupts the CPU
f) Finally ADC driver wakes up all user space applications and they can read data from shared memory

Comparison of the DMA with different memory allocation scheme

DMA with streaming mapped buffer usage.

Advantages

- Low memory usage
- No permanent mapping

Disadvantages

- Several data user problem
- High CPU usage for data pushing from kernel buffer to user space buffer
- Strong dependence of CPU usage and performance on number of user space applications which request data

DMA with coherent mapped buffer usage.

Advantages

- Low CPU usage and high performance
- Multiple users easy to implement

Disadvantages

- Permanent memory allocation for DMA
- Permanent mapping of this memory

Summary

- It is beneficial to have good MTCA general purpose driver
- Most functionalities are doable because of MTCA standards
- A lot of functionalities have already been implemented
- The driver is already being used for some devices
- A lot of drivers have been using some common functionalities from this driver (using driver stacking)

Outlook

The codes and current ready documentation are public and one can find them in DESY SVN public repository
https://svnsrv.desy.de/websvn/wsvn/General.ers/sandbox/drivers/general_driver
Any remarks are welcome. We are open for collaboration. If you are interested, you are welcomed to join the project.
If you have any questions, please contact
Davit Kalantaryan: davit.kalantaryan@desy.de
Ludwig Petrosyan: ludwig.petrosyan@desy.de

Thank you for your attention!