# Status of MTCA Driver Development at DESY in Zeuthen

## 3rd MTCA Workshop for Industry and Research

Davit Kalantaryan
3rd MTCA Workshop
DESY, 11.12.2014

HELMHOLTZ
ASSOCIATION

DESY

# Outline

- Introduction (MTCA driver development activity in Zeuthen)
- Generalized functionalities for driver
- Gaining/Releasing access to the device
- Several read/write/bwrite without intermediate execution
- Synchronization mechanisms
- Event based notifications of changing registers content

# Introduction

- The photo injector test facility at DESY, Zeuthen site (PITZ) is optimizing the electron source for the European XFEL.
- The MicroTCA system is also planned to be used at PITZ in order to have hardware/software configuration of different subsystems as close as possible to the European XFEL.
- At PITZ, it is not always possible to find the driver for MicroTCA devices, especially for homemade devices and sometimes additional work should be performed (special drivers have to be written) to adapt the system for PITZ. One of the examples is the PITZ timing system.
- The aim was to have a driver that is able to handle different MicroTCA devices without slowing down the overall performance.
- General purpose driver and corresponding user space API to work with mentioned driver were developed at DESY in Zeuthen.
- The driver is already in use for PITZ timing devices.
- Svn rep.: "https://svnsrv.desy.de/websvn/wsvn/General.ers/sandbox/drivers/general_driver"

# Generalized functionalities for driver

The driver is able to handle different MTCA devices because it generalizes functionalities those very often force driver authors to do specific device implementations.

Following is the list of generalized functionalities

1. Read/write arbitrary amount of device IO registers in atomic manner (AM) using read/write system calls.
2. Bitwise write (bwrite) arbitrary amount of bits from arbitrary amount of registers in AM using write system call.
3. Sequential read/write/bwrite from/to device IO registers in AM without intermediate calculations (or executions) using ioctl system call (Explanation of intermediate calculations on following slides).
4. Some other common IOCTLs (asking slot number, vendor id, …).
5. Sequential read/write/bwrite/ioctl with intermediate calculations (development in progress).
6. If one of the existing interrupt handling scheme (implemented for this driver) is sufficient for a device, then handling of such kind of interruptible device also can be done by the same driver.
7. Event based notifications on any change of device registers contents.

# Several read/write/bwrite without intermediate calculations

Devices can be controlled using their IO registers. Very often register access is more than one at once. As an example in the case of timer device for setting time delay there are registers corresponding to time base and value.

**Register for time base**
0 – Milliseconds
1 – Seconds

**Register for time value**

This kind of sequential register accesses take most of the cases when device specific ioctl is implemented and using general driver become not available.
There is ioctl call, that confirms arbitrary amount of IO operations in sequence.
Code example how to confirm this ioctl call is shown in the next slide.

**Prog. A) Setting time to 1 second**

1. **Set(0x00,1);** - setting time base to s
2. **Set(0x02,1);** - setting time

**Prog. B) Setting time to 900 milliseconds**

1. **Set(0x00,0);** - setting time base to ms
2. **Set(0x02,900);**- setting time

**Possible sequence of actions performed by two concurrent programs**

1. B) **Set(0x00,0);** // In this case finally time will be 900s instead of being ~1s
2. A) **Set(0x00,1);**
3. A) **Set(0x02,1);** // this may lead to serious problems in system
4. B) **Set(0x02,900);**

# Program that demonstrates PCIE_GEN_IOC_RW ioctl

```
#include "pciegen_io.h"
int main(int argc, char* argv[])
{
    short vsnValueW[]= {0xbbbb,0x0abc}; int nValueR;
    short vsnMasks[] = {0x000f,0x00bd};
    struct str_for_ioc_rw aIocRW;
    struct device_rw2 vRWStruct[2];/* memset(vRWStruct,0,2*sizeof(device_ioc_rw2));*/
    const char* cpcFlName = (argc>1)?argv[1]:"/dev/pcie_gens4";
    int nFd = open(cpcFlName,O_RDWR);
    if(nFd<0)
    {
        fprintf(stderr,"\"%s\" is wrong\n", cpcFlName);
        return 1;
    }
    /*1.  Preparing first operation*/
    vRWStruct[0].offset_rw = 0;
    /* W_D8, W_D16, W_D32,W_D8_BWISE,W_D16_BWISE,W_D32_BWISE (for write)*/
    vRWStruct[0].mode_rw = W_D16_BWISE;
    vRWStruct[0].barx_rw = 3; vRWStruct[0].count_rw = 2;
    vRWStruct[0].dataPtr = (u_int64_t)vsnValueW;
    vRWStruct[0].maskPtr = (u_int64_t)vsnMasks;
    /*2.  Preparing second operation*/
    vRWStruct[1].offset_rw = 0x2;
    /* RW_D8, RW_D16, RW_D32 (for read)*/
    vRWStruct[1].mode_rw = RW_D32;
    vRWStruct[1].barx_rw = 2; vRWStruct[1].count_rw = 1;
    vRWStruct[1].dataPtr = (u_int64_t)&nValueR;

    aIocRW.m_nIterations = 2;
    aIocRW.dataPtr = (u_int64_t)vRWStruct;

    ioctl(nFd,PCIE_GEN_IOC_RW,&aIocRW);
    close(nFd);
    return 0;
}
```

This ioctl call does arbitrary amount of R/W from/to device registers by sequence in atomic manner.

```
struct str_for_ioc_rw {
    u_int64_t dataPtr;
    int32_t    m_nIteration;
    char       m_reserved[sizeof(u_int64)-size
};
```

```
/* generic register access */
typedef struct device_rw2
{
    u_int      offset_rw; /* offset in address
    u_int      mode_rw; /* mode of rw (RW_D8
    u_int      barx_rw;  /* BARx (0, 1, 2, 3, 4, 5
    int32_t    count_rw;/* Not mandatory  */
    u_int64_t dataPtr;   /* pointer to data   */
    u_int64_t maskPtr; /* pointer to mask    */
    int16_t    lockBar; /* <0: all bars   */
    int16_t    lockFrom; /* <=0: from start   */
    int32_t    lockTo; /* <0: to the end   */
}device_rw2;
```

# Several IO operations with intermediate calculations

In the case of several dependent IO operations each register access depends on the results 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(0x00);** - set time base
2. **setValue = timeBase**==0 ? 1000 : 1;
3. **Set(0x02,setValue);** - setting time

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

1. **Set(0x00,1);** - setting time base to s
2. **Set(0x02,1);** - setting time

**Possible sequence of actions performed by two concurrent programs**

1. Initial time base = 0;
2. A) **timeBase = Get(0x00);** **(timeBase=0) => 1000 will be set**
3. B) **Set(0x02,1000);**
4. B) **Set(0x00,1);**
5. A) **(timeBase=0) => 1000 will be set** **// Intermediate calculation**
6. A) **Set(0x02,1000);** // In this case final time will be 1000s instead of being 1s

# Many operations in sequence with intermediate calculations

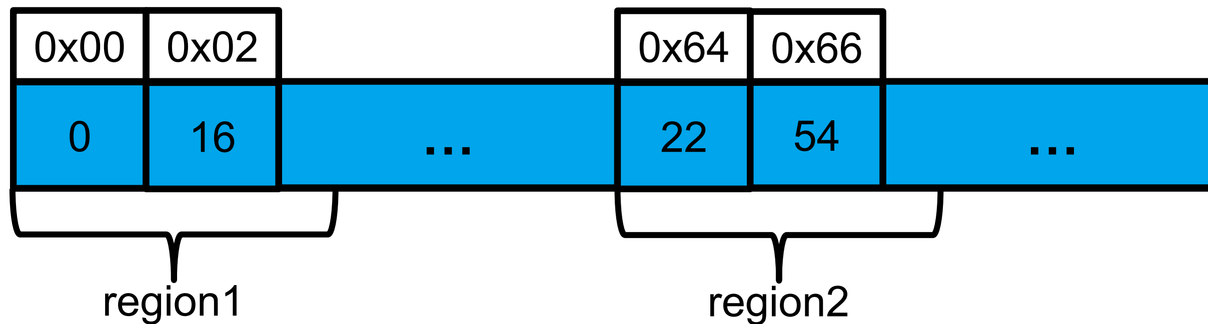
For implementing sequential accesses with intermediate calculations, two ioctl calls are implemented for both locking and unlocking. Between locking and unlocking the program can make any amount of system calls to driver for accessing device. Driver checks if the PID of program corresponds to the PID of locker program 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(PCIE_LOCK_REGION,aRegion);
2. read(), some calculations, write(), etc. …
3. ioctl(PCIE_UNLOCK_REGION);

**Whenever it is possible ioctl(PCIE_IOC_RW,…) 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.**

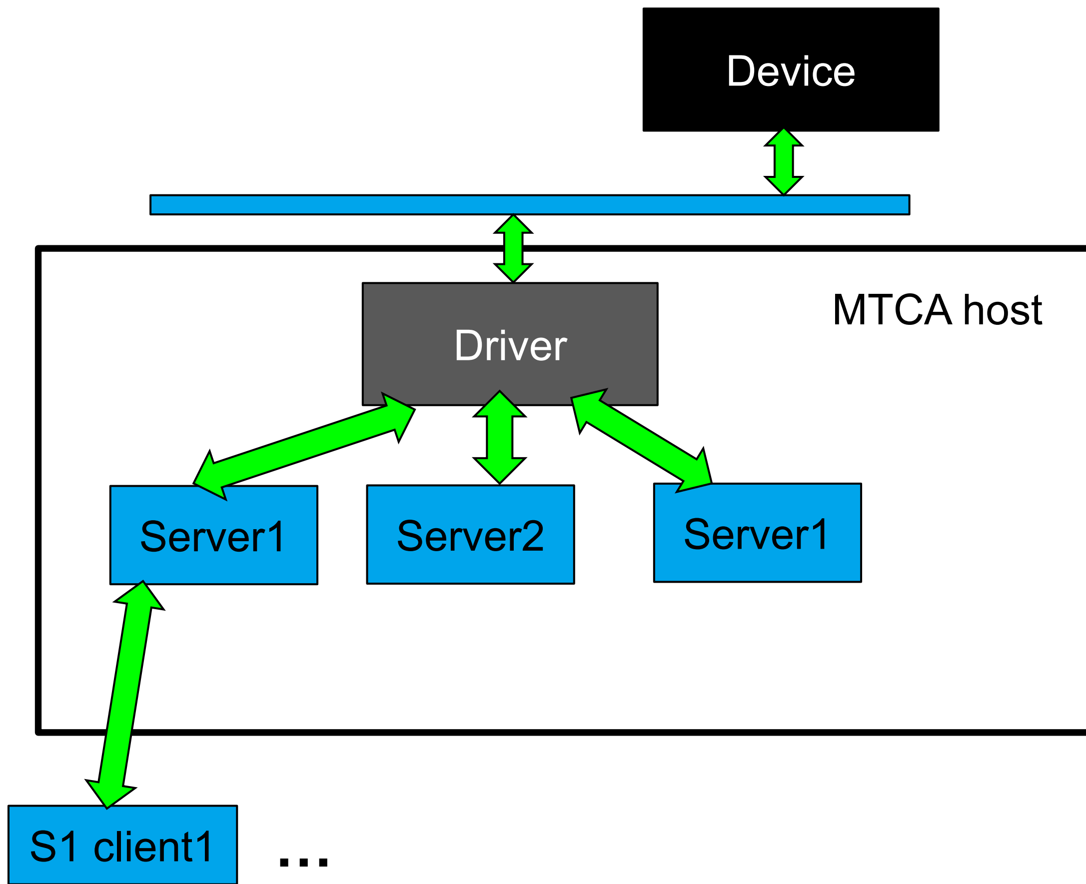
# Synchronization mechanisms

- Currently mutex is used to synchronize concurrent access to device.
- Development is in progress to change mutex with semaphore for making available several IO with intermediate calculations.




In these cases there is one synchronization object for whole device. In general there are regions those are not correlated to each other. In this case, it is not optimal to lock the device access to second region while working in the first region.
There are plans to use finally so called region locks. This means new synchronization object should have functionality to lock only specified regions.

# Event based notifications on changes in registers content




In the case of our timing system, 3 servers are running for one AMC device. Assume that each server has 5 clients and each client is asking for 10 different register contents (10 DOOCS properties) with 10 Hz. So number of IO access in one second will be **3x5x10x10=1500**.

By using ioctl call program can be registered in driver side for receiving a notification on any change in register content. Notification is an interrupt signal with information about region of registers those are changed. This information is in the 64 bit union field of si_value of structure siginfo_t.

# Some software packets for working with driver

- pcie_server (if this server runs on MTCA host, then it makes available all read/write/bwrite/ioctl operations from remote hosts)
- class PcieLocal (Provides interface for working with driver locally).
- class PcieRemote (Provides same interface as class PcieLocal for working with driver from remote host if pcie_server runs on driver host).
- pcie_tool (QT based GUI to communicate with local and remote hosts drivers. Compiled on different distributions of LINUX and WINDOWS7)
- pcie_tool_w (MFC based GUI, for remote communication from any WINDOWS hosts to the drivers, using pcie_server)

# Scheme of working remote programs

Device

MTCA host

Driver

pcie_server

Loc. prog1

Loc. prog2

DOOCS server

tcp

tcp

Rem. prog1

Rem. prog2

Local and remote configurator

mtcapitzcpu3:/dev/tamc200s4a

File Actions Help

168. Incoming signal 10, callback = 63505, si_value.sival_int = 533431442
169. Incoming signal 10, callback = 63505, si_value.sival_int = 533431443
170. Incoming signal 10, callback = 63505, si_value.sival_int = 533431444
171. Incoming signal 10, callback = 63505, si_value.sival_int = 533431445
172. Incoming signal 10, callback = 63505, si_value.sival_int = 533431446

2 Byte | Bar 0 | Hexadecimal

Address (Hex) 0

Number to RW 4

Read All | Write All

Data int Hexadecimal format

Addr. 0x4 Data 9776 Mask ffff R. W.

Addr. 0x6 Data b18b Mask ffff R. W.

Remote device register configurator

Dialog

Connected!
Device is opened

Host name or IP: mtcapitzcpu3

Disconnect

Device name: /dev/tamc200s4a

Close Device

Request IRQ

Gain access

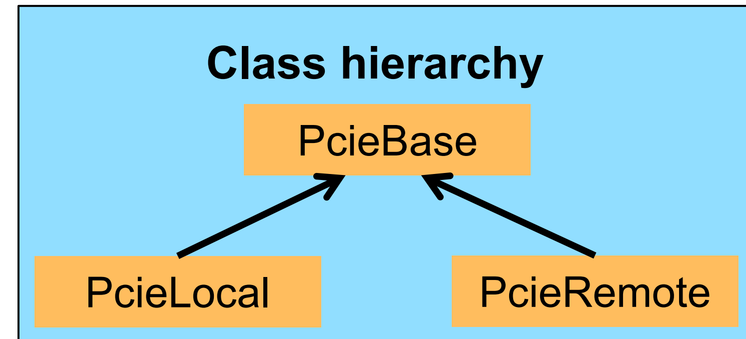
Mode 2 Byte | Bar Bar 0 | Hexadecimal

Addr. [Hex] 0 Data B70B Mask FFFF R. W.

OK | Cancel

All read/write/bwrite/ioctl are available from remote programs

# Some Interface Functions for C++ classes




```
class PcieBase
{
public:
    enum _TYPES_{REMOOTE,LOCAL};
    virtual             ~PcieBase(){}
    virtual int         GetType()=0;
                                   // For remote hostname:deviceentry
    virtual int         OpenDevice( const char* deviceName, int mode=O_RDWR )=0;
    virtual void        CloseDevice(  )=0;
    virtual int         RegisterForInterupt(int sigNum, int callback)=0;
    virtual void        UnRegisterFromInterupt()=0;
    virtual int         Read(  void* pBuffer, unsigned long offset, int mode, int count, int bar )const=0;
    virtual int         Write( void* cpBuffer, unsigned long offset, int mode, int count, int bar, void* cpMask = 0 )=0;
    virtual int         Iocntrl( unsigned long command, void* pDataInp=NULL, int InpDataLen=0,
                                 void* pDataOut=NULL, int* pOutDataLen=NULL )=0;

};
```

# Acknowledgment

Thanks to the Hamburg colleagues for big support on configuring and installing MTCAs:

V. Petrosyan → explanations how to prepare MTCA, configuration of MCH and providing management software and help to install them.
L. Petrosyan → help to understand driver development concepts and provide many code examples

Thank you for your attention!