low-cbf-mcs Documentation

CSIRO

Mar 29, 2021

Contents:

These will interact with higher levels of the SKA control system (i.e. Low.CSP) All require the SKA LMC base classes, which will be installed into the Docker image as part of the build process.

CHAPTER 1

LowCbfSubarray

CHAPTER 2

LowCbfCapLogicalStation

CHAPTER 3

LowCbfCapSearchBeam

CHAPTER 4

LowCbfCapStationBeam

CHAPTER 5

LowCbfCapTimingBeam

CHAPTER 6

AlveoDevice

AlveoDevice is a Tango device server for monitoring health status of Xilinx Alveo FPGA accelerator cards.

6.1 Usage

The *pciPath* property must be populated for each Tango device instance.

Use the full path to the *sysfs* location for user monitoring of the card. This will probably end in **00.1**, and will contain further subdirectories such as **xmc.u.<number>** and **rom.u.0**.

For example, in our test server we use: **/sys/devices/pci0000:ae/0000:ae:00.0/0000:af:00.1**

Note there is also a 'management' device (ends in **00.0**, contains **xmc.m.<number>**), it *probably* doesn't matter which you use but we have not tested this.

6.2 Testing

To test the software without using a real FPGA, you can use a copy of the *sysfs* files. There is one such set in **test-harness/alveo_sysfs**.

If you're running the device on a full Tango system, the *pciPath* parameter can be set to any filesystem location that is convenient to work with.

If you're only running a DeviceTestContext (or similar), you can set the path on the module itself, e.g.

```
from ska.low_cbf_mcs import AlveoDevice
AlveoDevice.pciPath = '/build/alveo_sysfs'
```

We have tested this using a Xilinx Alveo U50 Data Center Acclerator Card. We expect it to work with other cards in the range, but modifications to the monitored parameters are likely to be required to match the sensors present on each card.

6.3 Continuous Integration

The CI tests in this repo use DeviceTestContext and do not require a full Tango system. The files in **test-harness/alveo_sysfs** are used, which have known values that are hard-coded into the test script.

We had problems with the tests running very slowly and often failing, so as a temporary workaround we only run one test in the CI pipeline. There's a block of commented-out pytest parameters that should be reinstated once this issue is resolved. See **tests/test_alveo.py**.

CHAPTER 7

LowCbfFpga

LowCbfFpga is a Tango device server for monitoring and control of registers in the Low.CBF signal processing FPGAs. It's a lightweight wrapper around an AlveoCL "core" object, which communicates to the Alveo FPGA modules using PyOpenCL.

7.1 Simulation Mode

At present, the simulationMode attribute is read-only, and reflects whether the AlveoCL core failed to be instantiated. That is, "False" implies a connection to an FPGA.

Upon failure to create its AlveoCL core, the Tango device will create a dummy core. The dummy core handles reads & writes of attributes, obviously using the PC memory and no interface to any FPGA.

7.2 Continuous Integration

The current CI tests of this device use DeviceTestContext and do not require a full Tango system.

CHAPTER 8

AlveoCL

Instatiating an AlveoCL object requires arguments for firmware, logger, memory, and card number.

- Firmware is the path to an xclbin file.
- Logger should hopefully be compatible with a standard python logger - it just needs to be an object with .debug, .info, etc methods.
- Memories have a size in bytes, and a boolean "shared" flag (False means the memory is FPGA-only, True means shared with the PC).

Memory config must match the parameters of the firmware kernel.

Example:

```
from ska.low_cbf_mcs.alveo_cl import AlveoCL, MemConfig
memories = [
    MemConfig(1024 * 4, True), # 1024 words for register interchange
    MemConfig(1 << 30, True), # 1GiB
    MemConfig(128 << 20, True), # 128MiB
]
fpga = AlveoCL(
    "/app/my_kernel.xclbin", device.logger, mem_config=memories
)
```

Registers system.args_magic_number and system.fpga_uptime are created at object instantiation, but fpgamap is not parsed until start is called. (At present, a hard-coded filter selects only the 'system' and 'packetiser' peripherals)

```
fpga.start()
```

Reading & writing to FPGA registers is then performed using nested array syntax. Reads return an FpgaRegister, which contains value, time, etc.

```
# read example
print("Packets transmitted", fpga["packetiser"]["eth100g_tx_total_packets"].value)
```

(continues on next page)

(continued from previous page)

```
# write example
fpga["packetiser"]["psr_ctrl_vector"] = 3
```

CHAPTER 9

fpga_cmdline.py

A command-line (i.e. non-Tango) demonstration of OpenCL FPGA communications.

9.1 Command-Line Arguments

-m <size><k|M|G><s|i>

Configure memory buffers. Numeric size, followed by scale (k=1024, M=k*1024, G=M*1024), then 's' for shared between FPGA and host or 'i' for internal to FPGA. Multiple buffers may be specified, seperated by spaces or colons.

-f <path>

Path to the firmware (xclbin) file

-d <card number>

FPGA register addresses are hard-coded at present.

e.g.

```
python3 src/ska/low_cbf_mcs/fpga_cmdline.py -m "1Gs 128Ms" -f Packetiser_Dec_2020/
↪current_u50LV.xclbin -d 8
```

Press H in the CLI for further help.

CHAPTER 10

I2cDevice

For interfacing with the Gemini FPGA card backplane.

Other related Tango devices are listed on SKA Confluence - Perentie Tango Device Servers

CHAPTER 11

Indices and tables

- genindex
- modindex
- search