hyperfpga-comutils — User Guide

Function-by-function reference for the `hyperfpga_comutils` Python API (and the C API it mirrors).

```
from hyperfpga_comutils import Comblock, AXI_GPIO, AXI_DMA, UDMABuffer,
HlsControl, HardwareManager
```

1. Numeric formats

Every peripheral wrapper (`Comblock`, `AXI_GPIO`, `UDMABuffer`, `HlsControl`) is constructed with a `format_in` (used by `write_*`) and an optional `format_out` (used by `read_*`, defaults to `format_in`). Every `write_*`/`read_*` method also accepts a per-call `format=` that overrides the constructed default for that one call only.

Format string(s)	Width
`unsigned8` / `u8` / `uint8`	1 byte
`unsigned16` / `u16` / `uint16`	2 bytes
`unsigned32` / `u32` / `uint32` (default)	4 bytes
`unsigned64` / `u64` / `uint64`	8 bytes
`signed8` / `i8` / `int8`	1 byte
`signed16` / `i16` / `int16`	2 bytes
`signed32` / `i32` / `int32`	4 bytes
`signed64` / `i64` / `int64`	8 bytes
`float16` / `f16` / `half`	2 bytes
`float32` / `f32` / `float`	4 bytes
`float64` / `f64` / `double`	8 bytes
`fixedpoint` / `fixed` / `q`	`ceil((int_bits+frac_bits)/8)` bytes

Format strings are case-insensitive.

Fixed-point (Q-format): configured via the constructor's `int_bits` (default 8), `frac_bits` (default 24), and `signed` (default `True`) — i.e. Q8.24 by default. Encoding scales the float by `2**frac_bits`, rounds to the nearest integer, and packs it two's-complement (if signed) into `ceil((int_bits+frac_bits)/8)` bytes. Decoding does the inverse.

Overflow behavior: encoding raises (never silently saturates or wraps) when the value doesn't fit the target format — see §8 Error handling.

```
# Q1.31 fixed-point (full range [-1, 1))
cb = Comblock(comblock_num=0, default_format="fixedpoint", int_bits=1,
frac_bits=31, signed=True)
cb.write_reg(0, 0.5)
print(cb.read_reg(0))    # ~0.5

# Per-call override, independent of the object's configured default
cb.write_reg(1, 1234, format="unsigned32")
```

`AXI_GPIO.write_raw`, `read_raw` and `HlsControl.write_reg_raw`, `read_reg_raw` bypass this layer entirely: plain 32-bit values, no encode/decode.

2. `Comblock`

Interface to a ComBlock FPGA core through its Linux character devices (`/dev/ComBlock_<N>_regs_o`, `_regs_i`, `_fifo_o`, `_fifo_i`, `_ram`). Holds **no persistent OS resources** — it opens/closes a file descriptor per operation, so instances are cheap to create and safe to use from multiple ipyparallel engine calls.

Constructor

```
Comblock(
    comblock_num=0,
    default_format="unsigned32",
    int_bits=8, frac_bits=24, signed=True,
    format_out=None,
    fifo_batch_size=1024,
    reg_in_width=None, reg_out_width=None,
    fifo_in_width=None, fifo_out_width=None,
    ram_width=None,
)
```

Arg	Meaning
`comblock_num`	Device instance index (`ComBlock_<N>_*`).
`default_format`	Format string used by all `write_*` (and `read_*` unless `format_out` is given).
`int_bits`, `frac_bits`, `signed`	Fixed-point config, shared by both directions.
`format_out`	Separate format for `read_*` (e.g. write `float32`, read back `unsigned32`).
`fifo_batch_size`	Words per `write()`/`read()` syscall when streaming FIFO data (default 1024).

Arg	Meaning
`reg_in_width`, `reg_out_width`, `fifo_in_width`, `fifo_out_width`, `ram_width`	Override the data width (bits) for that channel instead of auto-detecting it from the devicetree (`REGS_IN_DWIDTH`, `REGS_OUT_DWIDTH`, `FIFO_IN_DWIDTH`, `FIFO_OUT_DWIDTH`, `DRAM_IO_DWIDTH`), which itself falls back to 32 if the devicetree property is absent.

Methods

Method	Description
`is_available() -> bool`	`True` if the ComBlock's register/FIFO device files exist.
`write_reg(register: int, value: float, format=None) -> None`	Write one AXI-Lite output register.
`read_reg(register: int, format=None) -> float`	Read one AXI-Lite input register.
`write_fifo(data, format=None) -> None`	Write a list/tuple/1-D NumPy (`float64` or `float32`) array to the output FIFO, batched per `fifo_batch_size`.
`read_fifo(length: int, format=None) -> np.ndarray[float64]`	Read `length` words from the input FIFO.
`fifo_in_clear() -> None`	Reset/drain the input FIFO.
`fifo_out_clear() -> None`	Reset the output FIFO.
`write_ram(addr: int, data: list[float], format=None) -> None`	Read-modify-write `data` into block RAM starting at word `addr` (4-byte aligned internally).
`read_ram(addr: int, length: int, format=None) -> np.ndarray[float64]`	Read `length` words from block RAM starting at `addr`.

Read-only properties

`reg_in_width`, `reg_out_width`, `fifo_in_width`, `fifo_out_width`, `ram_width` — resolved bit widths for each channel (constructor override → devicetree → 32-bit fallback).

Example

```
import numpy as np
from hyperfpga_comutils import Comblock

cb = Comblock(comblock_num=0)
cb.write_reg(0, 42.0)
print(cb.read_reg(0))
```

```
data = np.arange(100, dtype=np.float64)
cb.write_fifo(data)
rx = cb.read_fifo(100)

cb.write_ram(0, [1.0, 2.0, 3.0])
print(cb.read_ram(0, 3))
```

3. AXI_GPIO

Userspace driver for the Xilinx AXI GPIO IP, via the native Linux `gpiochip` character-device ABI (`/dev/gpiochipN`, backed by the in-tree `gpio-xilinx` driver — the same ABI `libgpiod` uses). Each pin is requested individually, so per-bit mixed input/output direction is supported. **Only channel 1** of the IP is exposed this way (see §9 for channel-2 access from C).

Constructor

```
AXI_GPIO(chip_path, format_in="unsigned32", format_out=None, int_bits=8,
frac_bits=24, signed=True)
```

`chip_path` is the gpiochip device, e.g. `/dev/gpiochip2`.

Methods

Method	Description
`set_direction(is_input: bool) -> None`	Set **all** pins to the same direction.
`set_direction_mask(mask: int) -> None`	Per-bit direction; bit `1` = input, `0` = output (matches the legacy TRI register convention).
`write(value: float, format=None) -> None`	Encode `value` per `format_in` and drive it onto the output-direction pins.
`read(format=None) -> float`	Read all pins and decode per `format_out`.
`write_raw(value: int) -> None`	Raw 32-bit write, no format conversion.
`read_raw() -> int`	Raw 32-bit read, no format conversion.
`set_pin(pin: int) -> None`	Set a single bit (read-modify-write).
`clear_pin(pin: int) -> None`	Clear a single bit.
`toggle_pin(pin: int) -> None`	Toggle a single bit.
`read_pin(pin: int) -> int`	Read a single bit (0/1).

Read-only property

`chip_path`.

Example

```
from hyperfpga_comutils import AXI_GPIO

gpio = AXI_GPIO("/dev/gpiochip2")
gpio.set_direction(False)   # all pins output
gpio.write(0xAA)
gpio.set_pin(3)
print(gpio.read_pin(3))
```

4. `AXI_DMA`

Userspace driver for the Xilinx AXI DMA IP in Simple/Direct-Register mode, via UIO (`/dev/uioN`) + mmap. Resets both channels on construction.

Constructor

```
AXI_DMA(uio_device)   # e.g. "/dev/uio1"
```

Methods

Method	Description
`start_mm2s(src_phys: int, length: int) -> None`	Kick off a Memory-Mapped→Stream transfer (non-blocking).
`wait_mm2s(timeout_s: float) -> None`	Block until MM2S goes idle or the timeout elapses; raises on bus error or timeout.
`start_s2mm(dst_phys: int, length: int) -> None`	Kick off a Stream→Memory-Mapped transfer (non-blocking).
`wait_s2mm(timeout_s: float) -> None`	Block until S2MM goes idle or the timeout elapses.
`transfer_mm2s(src_phys, length, timeout_s) -> None`	`start_mm2s` + `wait_mm2s` combined.
`transfer_s2mm(dst_phys, length, timeout_s) -> None`	`start_s2mm` + `wait_s2mm` combined.
`mm2s_status() -> int`	Raw MM2S `DMASR` register.
`s2mm_status() -> int`	Raw S2MM `DMASR` register.

Method	Description
`is_mm2s_idle() -> bool`	MM2S idle flag.
`is_s2mm_idle() -> bool`	S2MM idle flag.

`start_*`/`wait_*` poll the status register at ~1 ms intervals; there is no Python-exposed IRQ-driven variant (an IRQ-based path exists in the Rust core — `transfer_mm2s_irq`/`transfer_s2mm_irq` — but is not currently bound to Python or C).

Example — with `UDMABuffer` for streaming

```
from hyperfpga_comutils import UDMABuffer, AXI_DMA
import numpy as np

buf = UDMABuffer(device_num=0)
dma = AXI_DMA("/dev/uio1")

tx = np.linspace(1.0, 100.0, 1024, dtype=np.float32)
buf.write_numpy(tx, offset=0)
buf.sync_for_device(offset=0, size=1024 * 4)

dma.start_mm2s(buf.phys_addr, 1024 * 4)
dma.wait_mm2s(timeout_s=1.0)
```

5. `UDMABuffer`

Wraps a physically contiguous DDR buffer allocated by the `u-dma-buf` kernel driver (`/dev/udmabufN`), for use as a DMA source/destination. Access is byte-wise volatile (not `memcpy`) because the buffer is mapped as strict Device-type memory with no cache-coherent DT property — wide NEON/SIMD load-store instructions fault (SIGBUS) on that memory type.

Constructor

```
UDMABuffer(device_num=0, format_in="unsigned32", format_out=None,
int_bits=8, frac_bits=24, signed=True)
```

Methods

Method	Description
`write_bytes(data: bytes, offset: int) -> None`	Raw byte write at `offset`.
`read_bytes(offset: int, length: int) -> bytes`	Raw byte read.

Method	Description
`write_values(values: list[float], offset: int, format=None) -> int`	Encode each value per `format_in`/`format` and pack them back-to-back starting at `offset`. Returns the number of bytes written (handy for the DMA transfer `length` argument).
`read_values(count: int, offset: int, format=None) -> list[float]`	Read and decode `count` values starting at `offset`.
`write_numpy(array: np.ndarray[float32], offset: int) -> None`	Fast-path raw `float32` write from a 1-D NumPy array (bypasses the format layer).
`read_numpy(count: int, offset: int) -> np.ndarray[float32]`	Fast-path raw `float32` read.
`sync_for_device(offset: int, size: int) -> None`	Flush CPU writes so the FPGA sees them; `size=0` means "whole buffer from `offset`". Call **before** starting a DMA read of this region.
`sync_for_cpu(offset: int, size: int) -> None`	Invalidate/sync so the CPU sees FPGA writes; `size=0` means "whole buffer". Call **after** a DMA write into this region completes, before reading it back.

Read-only properties

`phys_addr` (physical address, for passing to `AXI_DMA.start_mm2s`/`start_s2mm`), `size` (buffer size in bytes).

Example

```
from hyperfpga_comutils import UDMABuffer

buf = UDMABuffer(0, format_in="signed32")
n_bytes = buf.write_values([1, -2, 3, 4], offset=0)
buf.sync_for_device(0, n_bytes)
# ... DMA transfer ...
buf.sync_for_cpu(0, n_bytes)
print(buf.read_values(4, 0))
```

6. `HlsControl`

Control-plane driver for a Vitis HLS IP block's `s_axi_CTRL`/`s_axi_CONTROL` interface (`ap_ctrl_hs` protocol: `ap_start`/`ap_done`/`ap_idle`/`ap_ready`/auto-restart). **Lifecycle control only** — scalar function arguments go through `Comblock`, and AXI4-Stream bulk data goes through

`AXI_DMA`/`UDMABuffer`. This split is deliberate: unlike ComBlock/GPIO/DMA, custom HLS IP has no shared devicetree `compatible` string, so it's identified by its Vivado instance name instead.

Constructor

```
HlsControl(uio_device, name="", format_in="unsigned32", format_out=None,
int_bits=8, frac_bits=24, signed=True)
```

Methods

Method	Description
`start() -> None`	Pulse `ap_start`, kicking off one execution.
`is_done() -> bool`	`ap_done` flag. Reading it via `wait_done` auto-clears it on most cores (`ap_ctrl_hs` semantics) — poll through `wait_done`, not by calling `is_done()` in a loop, if that matters for your core.
`is_idle() -> bool`	`ap_idle` flag.
`is_ready() -> bool`	`ap_ready` flag.
`set_auto_restart(enable: bool) -> None`	Core re-triggers itself on `ap_done` instead of going idle.
`wait_done(timeout_s: float) -> None`	Poll `ap_done` until set or timeout; raises on timeout.
`write_reg(offset: int, value: float, format=None) -> None`	Generic escape-hatch write at an arbitrary byte offset, through the format layer.
`read_reg(offset: int, format=None) -> float`	Generic escape-hatch read.
`write_reg_raw(offset: int, value: int) -> None`	Raw 32-bit write, no format conversion.
`read_reg_raw(offset: int) -> int`	Raw 32-bit read, no format conversion.

Read-only properties

`name` (Vivado instance name), `phys_addr`.

Example — HLS core + DMA-fed matrix multiply

```
import hyperfpga_comutils as hf

hw = hf.HardwareManager.detect()
```

```
hls = list(hw["hls_blocks"].values())[0]
dma = list(hw["dmas"].values())[0]
buf = hf.UDMABuffer(0)

tx_offset, rx_offset = 0, buf.size // 2
tx_len = buf.write_values(a_flat + b_flat, tx_offset, format="signed32")
buf.sync_for_device(tx_offset, tx_len)

# Arm the receive side before the core can emit output, then kick it,
# then feed input -- order matters for AXI-Stream cores with no FIFO slack.
dma.start_s2mm(buf.phys_addr + rx_offset, 64 * 4)
hls.start()
dma.start_mm2s(buf.phys_addr + tx_offset, tx_len)

dma.wait_mm2s(5.0)
dma.wait_s2mm(5.0)
hls.wait_done(2.0)

buf.sync_for_cpu(rx_offset, 64 * 4)
c_flat = buf.read_values(64, rx_offset, format="signed32")
```

See `tests/test_hls.py` (mockable on x86_64) and `tests/test_iphls.py` (full hardware run) for the runnable version.

7. `HardwareManager`

Static factory — no instantiation needed.

```
HardwareManager.detect() -> dict
```

Scans `/sys` and `/dev` at runtime instead of requiring hardcoded device paths, and returns:

Key	Value
`"comblocks"`	`{"comblock_<N>": Comblock(...), ...}`
`"gpios"`	`{"<gpiochip label>": AXI_GPIO(...), ...}`
`"dmas"`	`{"<devicetree name>": AXI_DMA(...), ...}`
`"hls_blocks"`	`{"<devicetree name>": HlsControl(...), ...}`
`"udmabuf_available"`	`bool`

Detection logic:

- **ComBlock** — probe `/dev/ComBlock_<N>_regs_o` for `N` in `[0, 16)`.
- **AXI GPIO** — scan `/sys/bus/gpio/devices/gpiochipN`, keep chips whose devicetree `compatible` contains `xps-gpio` (excludes the PS's own native `zynqmp-gpio*` controller, which shares the same bus).

- **AXI DMA** — scan `/sys/class/uio/`; a UIO node is AXI DMA if its devicetree node has a `dma-channel-mm2s` or `dma-channel-s2mm` child node. This is structural, not name-based, because a custom HLS IP can also have "dma" in its name (e.g. `matrix_dma`) without being one.
- **HLS block** — any other generic UIO peripheral, *except* one whose `name` sysfs attribute contains `axi-pmon` (the AXI Performance Monitor, which is skipped). This is exclusion-based best-effort, since custom HLS IP has no shared `compatible` string to match on positively.
- **udmabuf** — `/dev/udmabuf0` existing sets `udmabuf_available = True`.

Auto-detected `AXI_GPIO`/`HlsControl` instances default to `unsigned32` (raw passthrough), since detection has no per-instance format info to infer from — pass `format=` per call if you need a different encoding.

```
hw = HardwareManager.detect()
cb = hw["comblocks"]["comblock_0"]
cb.write_reg(0, 42.0)
```

8. Error handling

Every Rust-side error maps to a Python exception:

Situation	Python exception
A device file doesn't exist (`/dev/ComBlock_*`, `/dev/uio*`, `/dev/gpiochip*`, `/dev/udmabuf*`)	`FileNotFoundError`
FIFO not ready (empty on read / full on write, kernel `EAGAIN`) — caller should retry	`BlockingIOError`
DMA transfer timed out, or a DMA bus error was flagged in `DMASR`	`RuntimeError`
A value doesn't fit the selected numeric format (over/underflow), or an address is out of range	`ValueError`
Any other OS-level error (open/read/write/mmap/ioctl failure)	`OSError`

Encoding never silently saturates or wraps out-of-range values — it raises `ValueError` instead. Size your `int_bits`/`frac_bits` (fixed-point) or pick a wide-enough integer/float format accordingly.

```
try:
    cb.write_reg(0, 1e12, format="unsigned32")
except ValueError as e:
    print(f"value doesn't fit: {e}")
```

9. C API summary

`include/hyperfpga_comutils.h` + `libhyperfpga_comutils.so` expose a subset of the above as a plain C ABI, using opaque `*Handle` pointers and POSIX-style negative-errno return codes (`0` = success). Covered: `Comblock`, `AXI_GPIO` (legacy UIO/devmem backend — **not** the native `gpiochip` backend Python

uses, and it additionally exposes IP channel 2, which Python does not), `AXI_DMA`, `UDMABuffer` — all raw `uint32`/byte values only. **`HlsControl` and the numeric-format layer are Rust/Python only** and have no C binding.

Group	Functions
ComBlock	`comblock_open`, `comblock_close`, `comblock_write_reg`, `comblock_read_reg`, `comblock_write_fifo`, `comblock_read_fifo`, `comblock_fifo_in_clear`, `comblock_fifo_out_clear`
AXI GPIO (ch. 1)	`gpio_open_uio`, `gpio_close`, `gpio_set_direction`, `gpio_set_direction_mask`, `gpio_write`, `gpio_read`
AXI GPIO (ch. 2)	`gpio_set_direction_ch2`, `gpio_set_direction_mask_ch2`, `gpio_write_ch2`, `gpio_read_ch2`
AXI DMA	`dma_open_uio`, `dma_close`, `dma_start_mm2s`, `dma_wait_mm2s`, `dma_start_s2mm`, `dma_wait_s2mm`, `dma_transfer_mm2s`, `dma_transfer_s2mm`, `dma_is_mm2s_idle`, `dma_is_s2mm_idle`
UDMABuffer	`udmabuf_open`, `udmabuf_close`, `udmabuf_phys_addr`, `udmabuf_size`, `udmabuf_sync_for_device`, `udmabuf_sync_for_cpu`

```
struct hfpga_ComblockHandle *cb = NULL;
if (comblock_open(0, &cb) == 0) {
    comblock_write_reg(cb, 0, 12345);
    uint32_t val = 0;
    comblock_read_reg(cb, 0, &val);
    comblock_close(cb);
}
```

Error codes: `0` OK · `-2 ENOENT` (device not found) · `-5 EIO` (generic OS/I/O error) · `-11 EAGAIN` (FIFO not ready) · `-16 EBUSY` (DMA bus error) · `-62 ETIME` (DMA timeout) · `-22 EINVAL` (conversion/address error, or a null pointer argument).