GNU Radio 4
Block Development

Building an Out-of-Tree Module

A hands-on workshop for developers and engineers

Josh Morman, Altio Labs | GRCon26 Raleigh, NC

One module, six blocks

Lesson	Build	Learn
0	Empty scaffold	SDK, compiler, build
1 → 2	Gain → Gain<T>	Ports, SIMD, discovery
3 → 4	Square → bulk Square	Settings, graph execution, spans
5	Packetizer	PMT payloads and GR3
6	TagForwarder	Sample-aligned metadata
7	MovingAverage	History and buffer accounting
8	ThresholdDetector	Events and reset commands

Lesson 0: configure the scaffold

Activate the prebuilt SDK; select its compatible C++23 compiler.

```
source /path/to/gnuradio4/build/full/activate.sh
export CXX=/path/to/sdk-compatible/clang++
cmake -S . -B build -G Ninja \
  -DCMAKE_BUILD_TYPE=RelWithDebInfo \
  -DCMAKE_INSTALL_PREFIX="$GR4_PREFIX" \
  -DENABLE_TESTING=ON -DENABLE_EXAMPLES=ON
cmake --build build -j 2
ctest --test-dir build --output-on-failure
```

“No tests were found” is expected before Lesson 1.

Prerequisites and offline setup: docs/lesson00-setup.md

Work in one checkout

```
edit → configure / build → test → install → use
```

```
cmake -S . -B build
cmake --build build -j 2
ctest --test-dir build --output-on-failure
# From Lesson 2 onward, check the installed module too:
cmake --install build
```

Run from your working OOT root, in an activated SDK shell.

Where the work lives

```
blocks/tutorial/
  include/gnuradio-4.0/tutorial/   block headers
  test/                            QA tests
  examples/                        runnable C++ graphs

gr3_grc/                           GR3 receiver
solutions/lesson00/ … lesson08/     incremental overlays
test_external/consumer/            installed-package check
```

Start with empty header, test, and example lists. Add to them.

Public namespace: `gr::tutorial`

Resume at a completed lesson

From the repository root, prepare a **new directory**:

```
# Original processOne Square, ready for Lesson 4:
python3 scripts/prepare_lesson.py 3 build-lesson03

# Complete instructor reference:
python3 scripts/prepare_lesson.py 8 build-completed
```

Enter the chosen directory; configure, build, and test there.

The helper overlays lessons 00 through N. It refuses an existing destination.

First block: multiply by two

```
struct Gain : gr::Block<Gain> {
    using gr::Block<Gain>::Block;
    using Description = gr::Doc<"Multiply each input sample by two.">;

    gr::PortIn<float> in;
    gr::PortOut<float> out;
    float factor = 2.0f;
    GR_MAKE_REFLECTABLE(Gain, in, out);

    [[nodiscard]] constexpr float processOne(float input) const noexcept {
        return input * factor;
    }
};
```

Base · typed ports · reflection · `processOne`

```
GR_REGISTER_BLOCK(gr::tutorial::Gain)
```

GR4 supplies the loop

```
[[nodiscard]] constexpr float processOne(float input) const noexcept {
    return input * factor;
}
```

```
input:    1.0   −2.0   3.5
output:   2.0   −4.0   7.0
```

For simple one-in / one-out processing, **GR4 supplies the loop**.

LIVE: Build and test Gain

```
cmake --build build -j 2
ctest --test-dir build --output-on-failure -R qa_Gain
```

Check positive and negative inputs: **every output is doubled**.

Open Gain.hpp and qa_Gain.cpp; use the lesson01 versions.

Give the ports a sample type

Before: `Gain`

```
struct Gain : gr::Block<Gain> {
```

```
gr::PortIn<float> in;
gr::PortOut<float> out;
float factor = 2.0f;
```

After: `Gain<T>`

```
template<typename T>
struct Gain : gr::Block<Gain<T>> {
```

```
gr::PortIn<T> in;
gr::PortOut<T> out;
T factor = T{2};
```

Declaration excerpts · uses become `Gain<float>`.

Same algorithm, scalar or SIMD

```
template<gr::meta::t_or_simd<T> V>
[[nodiscard]] constexpr V processOne(const V& input) const noexcept {
    return static_cast<V>(input * factor);
}
```

```
using V = gr::meta::simdize<float>;
V input([](auto i) { return static_cast<float>(i) - 2.5f; });
const V output = gain.processOne(input);
for (std::size_t i = 0; i < V::size(); ++i)
    expect(eq(output[i], 2.0f * input[i]));
```

QA excerpt · different lane values; no fixed lane count or instruction promise.

C++ type → runtime name

C++	Registry
`Gain<float>`	`Gain<float32>`
`Gain<std::int16_t>`	`Gain<int16>`
`Gain<std::int32_t>`	`Gain<int32>`
`Gain<std::uint8_t>`	`Gain<uint8>`

All names above have the prefix `gr::tutorial::`.

Write `float` in C++; use `float32` in the runtime type ID.

Register concrete types

```
GR_REGISTER_BLOCK(gr::tutorial::Gain, [T], [float, std::int16_t, std::int32_t, std::uint8_t])
```

```
header list → registration generator → plugin library
```

The list determines what the runtime factory can create.

Keep the macro on one physical line.

LIVE: Install and open Studio

```
cmake --install build
./build/blocks/tutorial/examples/tutorial_plugin_check
gr4-studio
```

1. Search the catalog for **tutorial** or **Gain**.
2. Place the float variant; inspect `gr::tutorial::Gain<float32>`.
3. Reinstall and relaunch Studio after plugin changes.

Installed plugin → local server → `/blocks` → Studio catalog

A new block: Square

```
y = gain_linear × x²

gain_linear = 10^(gain_db / 20)
```

Setting	Input	Output
0 dB	`1, −2, 3.5`	`1, 4, 12.25`
20 dB	`1, −2, 3.5`	`10, 40, 122.5`

Float and complex float · complex square means `x * x`.

Expose a setting

```
gr::Annotated<float, "gain_db", gr::Doc<"Gain applied after squaring">,
              gr::Unit<"dB">> gain_db{0.0f};

GR_MAKE_REFLECTABLE(Square, in, out, gain_db);
```

```
void settingsChanged(const gr::property_map&,
                     const gr::property_map& new_settings) {
    if (new_settings.contains("gain_db")) {
        _gain_linear = std::pow(10.0f, gain_db.value / 20.0f);
    }
}
```

Value + documentation + units · reflection exposes the setting.

Cache the derived value

```
gain_db → settingsChanged() → _gain_linear
                                    ↓
                               processOne()
```

```cpp
[[nodiscard]] T processOne(T input) const noexcept {
    return _gain_linear * input * input;
}
```

```cpp
private:
    float _gain_linear{1.0f};
```

Settings change rarely; samples arrive constantly.

Apply settings through the framework

Direct test: initialize and apply the starting value.

```
Square<float> square(gr::property_map{{"gain_db", 20.0f}});
square.settings().init();
std::ignore = square.settings().applyStagedParameters();
```

Stage a change; verify it was accepted; apply it.

```
expect(square.settings().setStaged({{"gain_db", 0.0f}}).empty());
std::ignore = square.settings().applyStagedParameters();
expect(std::abs(square.processOne(-2.0f) - 4.0f) < 0.001f);
```

Test excerpts · a running graph applies staged changes at a processing boundary.

From a direct call to a running graph

```
finite source → Square (20 dB) → recording sink
  1, −2, 3.5                      10, 40, 122.5
```

```
gr::scheduler::Simple scheduler;
checked(scheduler.exchange(std::move(graph)));
checked(scheduler.runAndWait());
```

Create and configure → connect ports → transfer graph → run.

Execution excerpt · inspect the complete example in square_graph.cpp.

LIVE: Square settings in Studio

```
./build/blocks/tutorial/examples/square_graph
```

Expected: **10, 40, 122.5** with `gain_db = 20`.

> 1. Install, relaunch Studio, and place **Square<float32>**.
> 2. Inspect `gain_db`: default **0**; set the graph property to **20**.
> 3. Use the new property on the next graph run.

A saved graph property and a running session's setting are separate operations.

Two processing models

`processOne`	`processBulk`
One item in the function	Spans in the function
GR4 supplies the loop	Our function handles the chunk
Good fit for Gain	Useful for grouping and buffer APIs

When bulk helps: Packetizer groups N samples per input into one PMT;

a buffer based DSP library accepts spans in one call.

Square is our controlled comparison: **same DSP, new work function**.

Replace Square's earlier `processOne` with `processBulk`.

Square: replace the work function

```
[[nodiscard]] gr::work::Status processBulk(std::span<const T> input,
                                           std::span<T> output) const noexcept {
    std::ranges::transform(input, output.begin(), [this](T value) {
        return _gain_linear * value * value;
    });
    return gr::work::Status::OK;
}
```

`std::ranges::transform` applies the same calculation.

Full spans processed → automatic consumption and publication.

A work call has no signal meaning

```
logical signal:
 1  2  3  4  5  6  7  8

scheduler may give:
[1  2] [3  4  5] [6] [7  8]
```

work call ≠ frame · packet · signal boundary

Spans belong to this call. Do not retain their storage.

A stream can carry objects

```
sample streams → Packetizer → gr::pmt::Value
```

Numbers · strings · vectors · maps · nested values

This Packetizer puts a numeric `gr::Tensor<T>` inside each value.

PMT is a generic value representation. The output is still a typed stream port.

Packetizer: declare the interface

```
template<typename T>
struct Packetizer : gr::Block<Packetizer<T>, gr::Resampling<1024U, 1U, false>> {
    using Base = gr::Block<Packetizer<T>, gr::Resampling<1024U, 1U, false>>;
    using Base::Base;
    using Description = gr::Doc<"Sum input streams into PMT numeric vectors.">;

    std::vector<gr::PortIn<T>> in{2};
    gr::PortOut<gr::pmt::Value> out;
    gr::Annotated<gr::Size_t, "n_inputs", gr::Limits<1U, 8U>> n_inputs{2U};
    gr::Annotated<gr::Size_t, "packet_size", gr::Limits<1U, 1024U>> packet_size{1024U};

    GR_MAKE_REFLECTABLE(Packetizer, in, out, n_inputs, packet_size);
```

Declaration excerpt · `in` is a collection of typed ports.

N samples from each input → one object

```
void settingsChanged(const gr::property_map&,
                     const gr::property_map& new_settings) {
    if (new_settings.contains("n_inputs")) {
        in.resize(static_cast<std::size_t>(n_inputs.value));
    }
    if (new_settings.contains("packet_size")) {
        this->input_chunk_size = packet_size.value;
        this->output_chunk_size = 1U;
    }
}
```

`Resampling<1024U, 1U, false>`: default input count, output count, mutable sizes.

Configure ports and rate **before wiring the graph**.

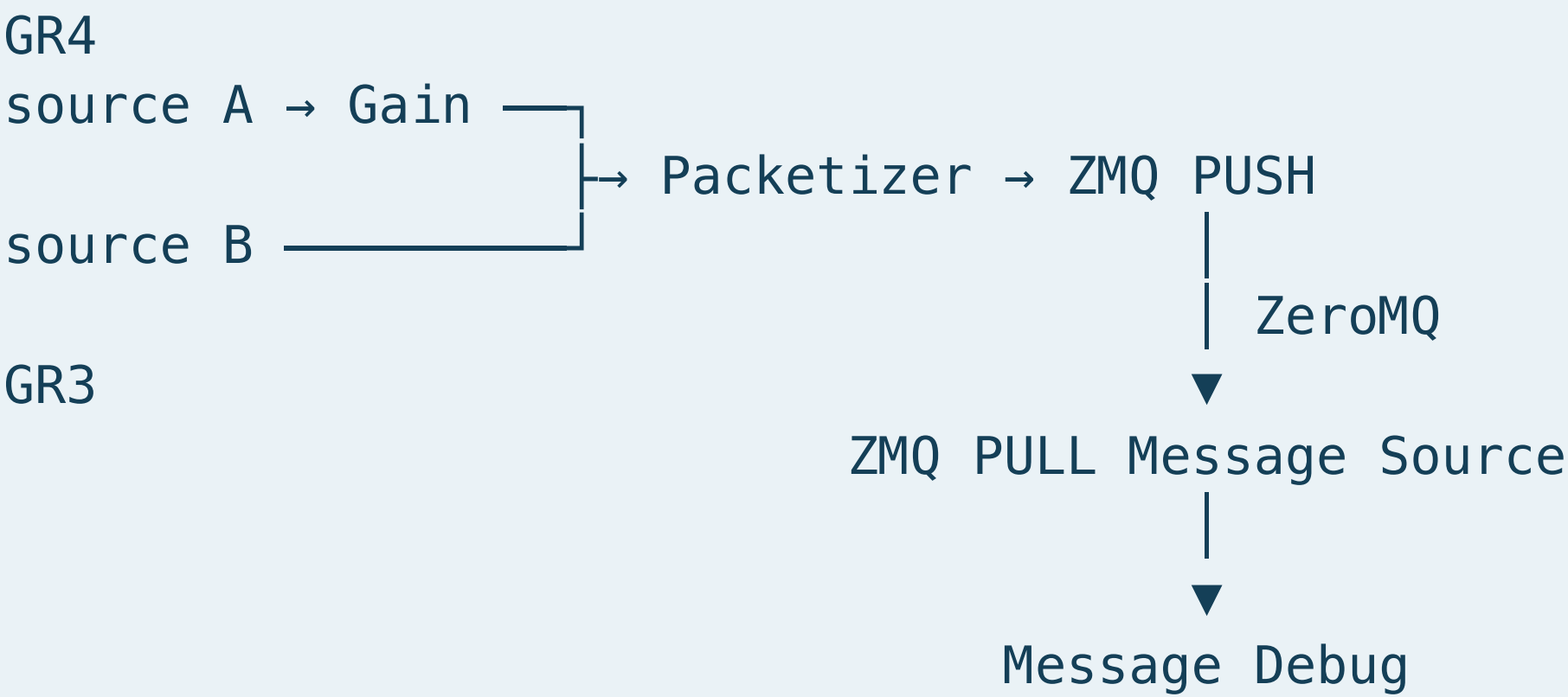
Complete groups only: drop a short final tail; no padding.

Sum the inputs; own the payload

```
template<gr::InputSpanLike TInSpan>
[[nodiscard]] gr::work::Status processBulk(const std::span<TInSpan>& inputs,
                                           std::span<gr::pmt::Value> output) const {
    const auto n = static_cast<std::size_t>(this->input_chunk_size.value);
    for (std::size_t packet = 0; packet < output.size(); ++packet) {
        const auto offset = packet * n;
        std::vector<T> sum(n);
        std::ranges::copy(std::span<const T>(inputs[0].data() + offset, n),
                          sum.begin());
        for (std::size_t channel = 1; channel < inputs.size(); ++channel) {
            const auto slice = std::span<const T>(inputs[channel].data() + offset, n);
            std::ranges::transform(sum, slice, sum.begin(), std::plus<T>{});
        }
        output[packet] = gr::pmt::Value(gr::Tensor<T>(gr::data_from, sum));
    }
    return gr::work::Status::OK;
}
```

`InputSpanLike` constrains each GR4 input span; the outer `std::span` holds one per port.

Keep the existing GR3 application



`tcp://127.0.0.1:5558` · GR4 binds · GR3 connects

Select the GR3 wire format explicitly

```
auto& sink = graph.emplaceBlock<zeromq::ZmqPushSink<gr::pmt::Value>>({
    {"endpoint", "tcp://127.0.0.1:5558"}, {"bind", true},
    {"timeout", 100}, {"pass_tags", false}, {"pmt_wire_format", "GR3"}});
```

```
Tensor<float> → GR3 PMT binary codec → f32 vector
```

The supplied SDK defaults to **`GR4_YAML_V1`**.

The GR3 receiver needs **`pmt_wire_format = "GR3"`**.

Graph excerpt · GR3 uses a PULL Message Source.

LIVE: GR4 → GR3 over ZeroMQ

Terminal A · activated SDK

```
./build/blocks/tutorial/examples/zmq_packetizer
```

Terminal B · GR3 Python bindings available

```
python3 gr3_grc/zmq_packet_receiver.py
```

Four alternating f32 vectors → receiver prints **PASS** and exits.

Stop the publisher with Ctrl-C after receipt.

The data made it through

```
[12, 24, 36]
[48, 60, 72]
[12, 24, 36]
[48, 60, 72]
```

Legacy GR3 PMT codec → bare f32 uniform vectors

Validated values; receiver may start on either vector.

PMT and Packet have different uses

Representation	Good fit
`gr::pmt::Value`	Generic structured values
`gr::Packet<T>`	Typed packet-like signal data

Packet has typed signal values, a timestamp, and metadata fields.

We use PMT here for the **generic payload and GR3 interface**.

A tag adds a stream position

```
sample index:  0   1   2   3   4   5
sample value: 10  11  12  13  14  15
                          ↑
                {"tutorial.input": 42}
```

TagForwarder copies the samples, forwards the tag,
and adds one marker at the start of the run.

Copy samples; forward the tag maps

```
[[nodiscard]] gr::work::Status processBulk(gr::InputSpanLike auto& input,
                                           gr::OutputSpanLike auto& output) {
    std::ranges::copy(input, output.begin());
    if (!_marker_emitted && !output.empty()) {
        output.publishTag(gr::property_map{{"tutorial.marker", true}}, 0UZ);
        _marker_emitted = true;
    }
    for (const auto& [relative_index, map_ref] : input.tags(input.size())) {
        if (relative_index >= 0) {
            output.publishTag(map_ref.get(), static_cast<std::size_t>(relative_index));
        }
    }
    return gr::work::Status::OK;
}
```

`gr::NoTagPropagation` · `start()` re-arms the marker for each run.

Offsets are relative to the span

```
absolute input index:   0   1   2 | 3   4   5
current input span:                [13  14  15]
relative span index:                0   1   2
                                    ↑
                              tag at absolute 3
```

`input.tags(...)` reports **0** in this call.

`output.publishTag(map, 0)` puts it at this output span's start.

LIVE: Tag test

```
ctest --test-dir build --output-on-failure -R qa_TagForwarder
```

```
index 0: {"tutorial.marker": true}
index 3: {"tutorial.input": 42}
```

Tag: position + metadata · **Setting:** configuration

PMT payload: object contents

Now the output needs earlier samples

```
window_size = 3

input:    1  2  3  4  5  6
output:      2  3  4  5
```

MovingAverage waits for a **full window**.

No padding · window one is pass-through · input metadata is dropped.

Keep signal state in the block

```
std::vector<T> _history = std::vector<T>(3);
T _sum{};
std::size_t _position{0};
std::size_t _fill{0};
```

Member	Meaning
`_history`	Recent samples in a ring
`_sum`	Running sum
`_position`	Next ring slot
`_fill`	Samples accumulated so far

Replace the oldest sample

```
const T sample = input[n_consumed++];
if (_fill == window) {
    _sum -= _history[_position];
} else {
    ++_fill;
}
_history[_position] = sample;
_sum += sample;
_position = (_position + 1) % window;

if (_fill == window) {
    output[n_produced++] = _sum / static_cast<float>(window);
}
```

Loop-body excerpt · the full function checks output capacity before consuming.

Report only the work actually done

Window 3, fresh state	Consumed	Published
Input `[1, 2]`	2	0
Next input `[3, 4, 5]`	3	3 → `[2, 3, 4]`

```
const bool consumed = input.consume(n_consumed);
output.publish(n_produced);
return consumed ? gr::work::Status::OK : gr::work::Status::ERROR;
```

Check output capacity before consuming a sample that needs it.

Function-tail excerpt · `OK` reports status; span methods report counts.

LIVE: Same signal, arbitrary chunks

```
calls:     [1 2] [3 4 5] [6]
outputs:     []    [2 3 4] [5]

combined:   2 3 4 5
```

```
ctest --test-dir build --output-on-failure -R qa_MovingAverage
```

Signal state belongs to the block, not the work call.

A window change starts fresh

```
void settingsChanged(const gr::property_map& old_settings,
                     const gr::property_map& new_settings) {
    if (new_settings.contains("window_size") &&
        old_settings.at("window_size") != new_settings.at("window_size")) {
        resetState();
    }
}
```

```
window 3 → window 2 → clear history

new input: [10] [20]     output: [] [15]
```

Same value preserves state. `start()` resets for a new run.

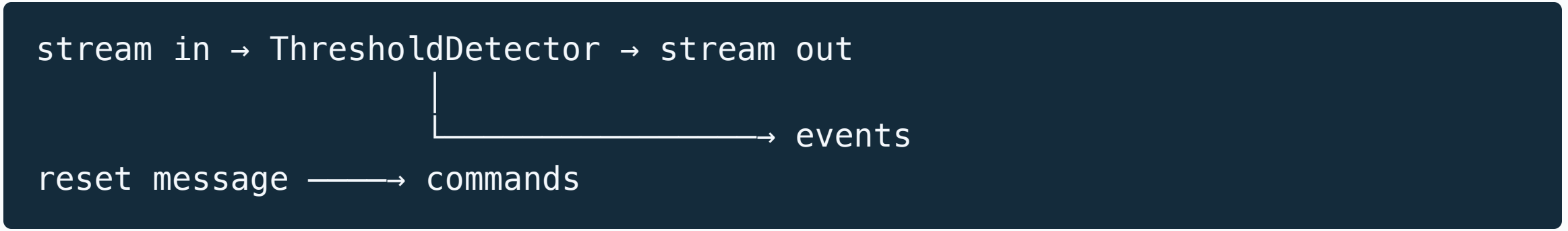
Four ways to interact with a block

Mechanism	Use
Stream	Data flow
Setting	Configuration
Tag	Metadata at a stream position
Message	Discrete event / command

PMT is a value representation, used where these mechanisms need it.

Detect crossings, keep the stream moving

```
threshold = 1.0
input:   0.2  0.7  1.1  1.3  0.8  1.4
events:             ↑              ↑
```



Two crossings → two events. Samples pass through unchanged.

Back to processOne: send an event

```
[[nodiscard]] float processOne(float input) {
    const bool above = input >= threshold.value;
    if (above && !_above) {
        gr::sendMessage<gr::message::Command::Notify>(
            events, this->unique_name, "threshold_crossing",
            gr::property_map{{"event", "threshold_crossing"}, {"value", input}});
    }
    _above = above;
    return input;
}
```

Method excerpt · `_above` is persistent, initialized to `false`.

Non-const scalar processing keeps the crossing state in order.

Application ports and the reset handler

```
gr::MsgPortIn commands;
gr::MsgPortOut events;
```

Set → `"reset"`

Successful empty body

Re-arm the detector.
Keep the threshold.

```
void processMessages(gr::MsgPortIn& /*port*/,
                     std::span<const gr::Message> messages) {
    for (const auto& message : messages) {
        if (!message.serviceName.empty() &&
            message.serviceName != this->name &&
            message.serviceName != this->unique_name) {
            continue;
        }
        if (message.cmd == gr::message::Command::Set &&
            message.endpoint == "reset" &&
            message.data.has_value() && message.data->empty()) {
            _above = false;
        }
    }
}
```

Declaration/method excerpts · reflect both ports; preserve the base `processMessages` overload.

LIVE: Observe events and reset

```
ctest --test-dir build --output-on-failure -R qa_ThresholdDetector
```

Stream unchanged · two `threshold_crossing` events: **1.1, 1.4**

```
1.1 → event    1.3 → no event
        reset command
1.4 → event    threshold still 1.0
```

What we've built

Block	Concept
Gain	Typed ports, scalar/SIMD, registration
Square	Settings, derived state, bulk processing
Packetizer	Dynamic inputs, rate contract, PMT / GR3
TagForwarder	Metadata at stream positions
MovingAverage	Persistent signal state
ThresholdDetector	Application events and commands

Verify the installed module

After the full build, CTest, and install, run the Lesson 8 checks:

```
./build/blocks/tutorial/examples/tutorial_plugin_check
cmake -S test_external/consumer -B build-consumer -G Ninja \
  -DCMAKE_CXX_COMPILER="$CXX"
cmake --build build-consumer -j 2
./build-consumer/tutorial_consumer
./build-consumer/tutorial_registry_consumer
python3 test_external/check_studio.py
```

9 CTest checks · 6 blocks · 12 registered variants

Repeat the GR3 receiver check. Rehearsal evidence: VALIDATION.md (2026-09-13).

Take these patterns into your OOT

- Build your own OOT with the supplied structure.
- Inspect `gnuradio4-blocks` for working examples.
- Use Studio to inspect installed blocks and settings.
- Extend one pattern; make its behavior testable.

Keep the **docs/ lessons**, the QA tests, and the lesson snapshots nearby.