Zephyr® Project

Developer Summit

Zephyr® Project
Developer Summit

High Bandwidth Sensors

Yuval Peress, *Google*

Email: peress@google.com
Discord: yuval#5515
GitHub: yperess

Agenda

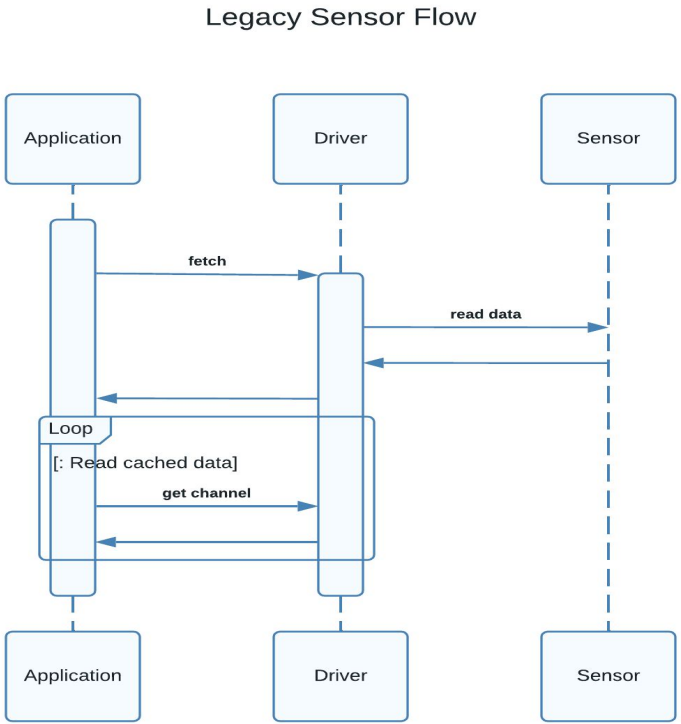
- What are high bandwidth sensors and why should you care?
- The evolution of sensors in Zephyr
- New feature: streaming sensor data

What are high bandwidth sensors?

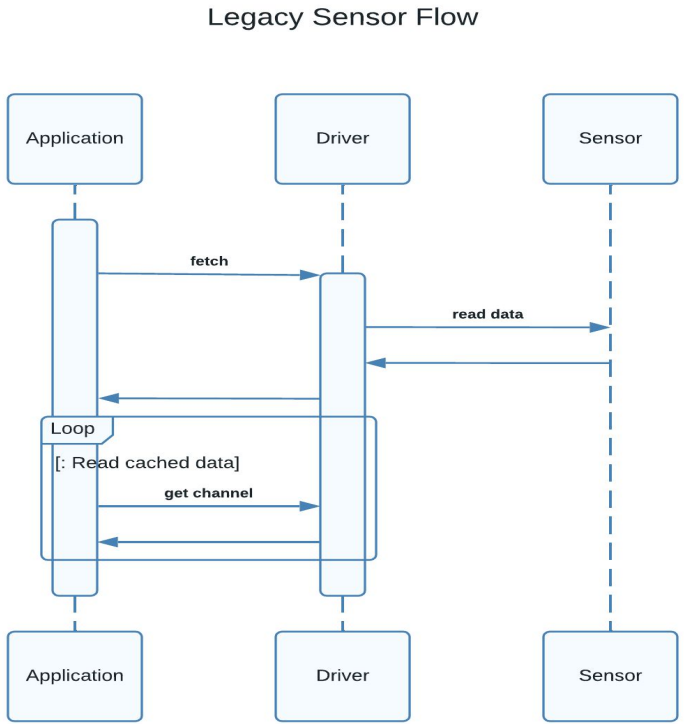
- Sensors that provide more than *X* samples/second?
- Sensors that provide more than *X* bytes/second?

Any sensor who's data pipeline is a bottleneck.

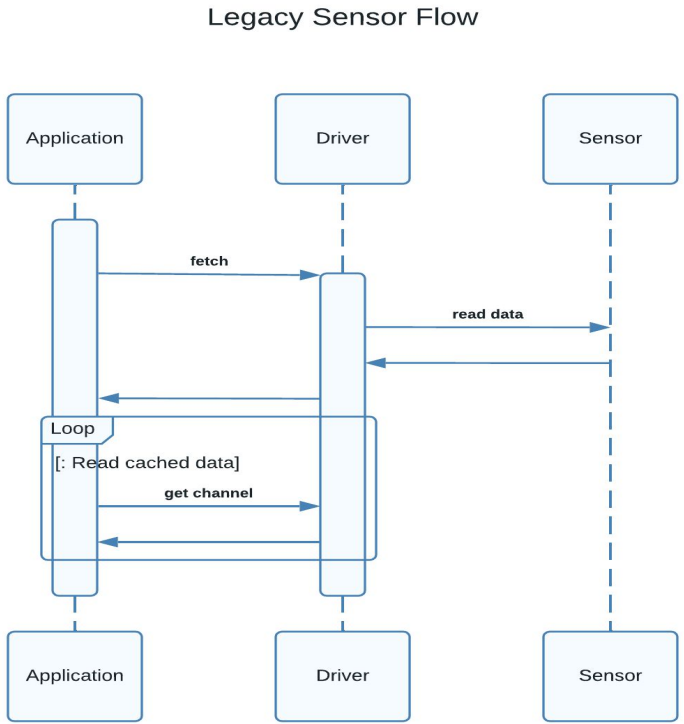
1. Application calls fetch



1. Application calls fetch
2. Driver performs bus transactions to get the data

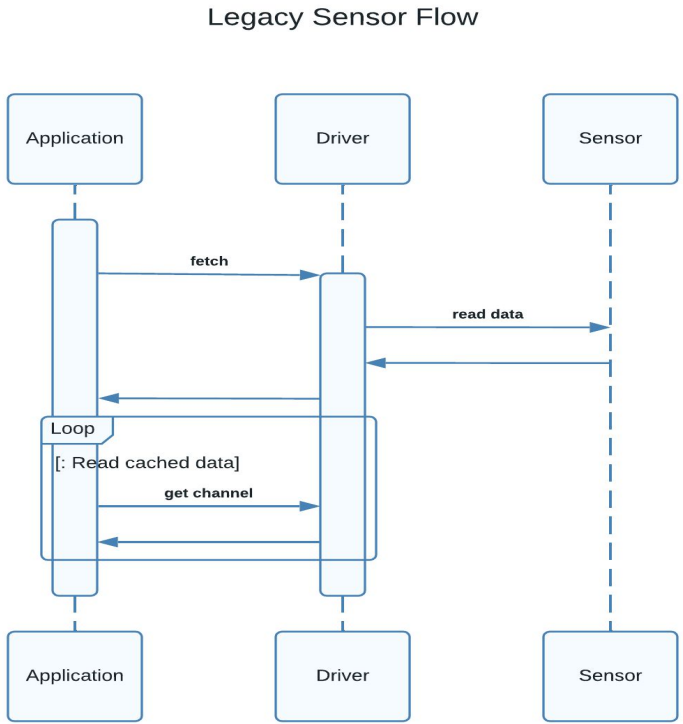


1. Application calls fetch
2. Driver performs bus transactions to get the data
3. Application reads channels from driver cache

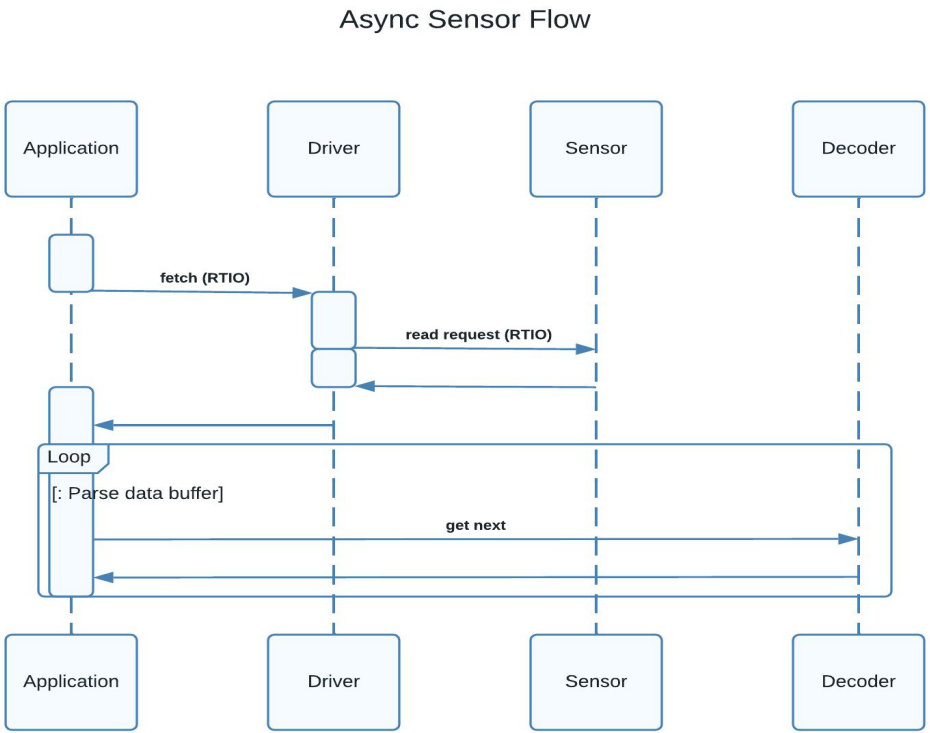


Problems:

- Application blocks during I/O
- Data processing assumes driver is locked
- Fixed memory owned by driver



- No blocking during bus I/O
- Data processing does not require driver owned cache
- Flexible memory use via mempool



Enabling the async API

```
# Add these to your prj.conf
CONFIG_SENSOR=y
CONFIG_SENSOR_ASYNC_API=y
```

One-shot data

Setting up the reader

```
SENSOR_DT_READ_IODEV(
    my_reader,              // IODev name
    DT_CHOSEN(lid_accel), // Sensor node to read
    SENSOR_CHAN_ACCEL_XYZ // One or more channels to read
);

RTIO_DEFINE_WITH_MEMPOOL(
    sensor_read_rtio, // RTIO name
    8,                // Submit queue size
    8,                // Completion queue size
    32,               // Number of memory blocks
    64,               // Block size (bytes)
    4                 // Block alignment (bytes)
);
```

Setting up the reader

```
SENSOR_DT_READ_IODEV(
    my_reader,              // IODev name
    DT_CHOSEN(lid_accel), // Sensor node to read
    SENSOR_CHAN_ACCEL_XYZ // One or more channels to read
);

RTIO_DEFINE_WITH_MEMPOOL(
    sensor_read_rtio, // RTIO name
    8,                // Submit queue size
    8,                // Completion queue size
    32,               // Number of memory blocks
    64,               // Block size (bytes)
    4                 // Block alignment (bytes)
);
```

Why use mempools?

- Allows us to delay processing
- Allows us to control how memory is managed
  - Small blocks for one-shot reading
  - Large blocks for streaming fast sensors
  - Lots of small blocks for mixed use

Queuing the read

```
int rc = sensor_read(
    &my_reader,
    &sensor_read_rtio,
    /*userdata=*/ my_reader.sensor
);

if (rc != 0) {
  LOG_ERR("Failed to initiate a read (%d)", rc);
}
```

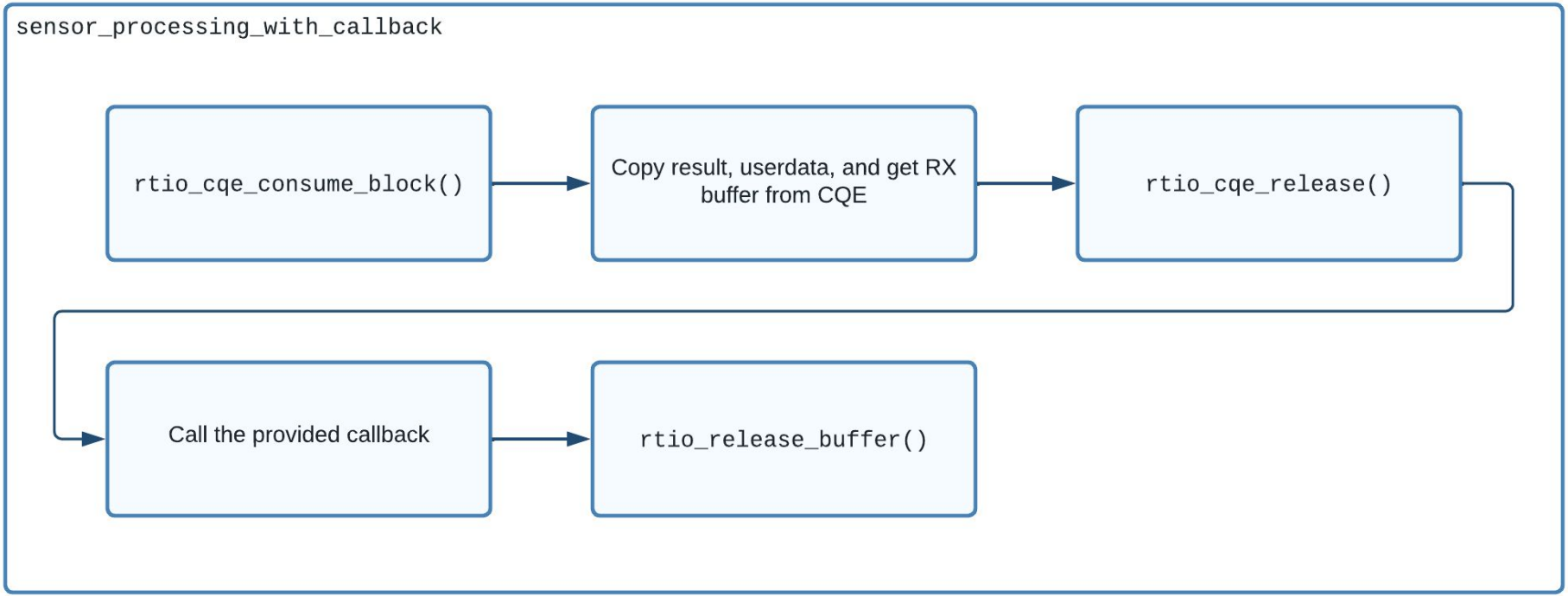
Processing the data

```
int rc = sensor_read(
    &my_reader,
    &sensor_read_rtio,
    /*userdata=*/ my_reader.sensor
);

if (rc != 0) {
  LOG_ERR("Failed to initiate a read (%d)", rc);
}

// Block until data is read, then call 'my_callback'
sensor_processing_with_callback(&sensor_read_rtio, my_callback);

// Callback signature:
// void (*)(int result, uint8_t *buf, uint32_t buf_len, void *userdata)
```


sensor_processing_with_callback
rtio_cqe_consume_block()
Copy result, userdata, and get RX buffer from CQE
rtio_cqe_release()
Call the provided callback
rtio_release_buffer()

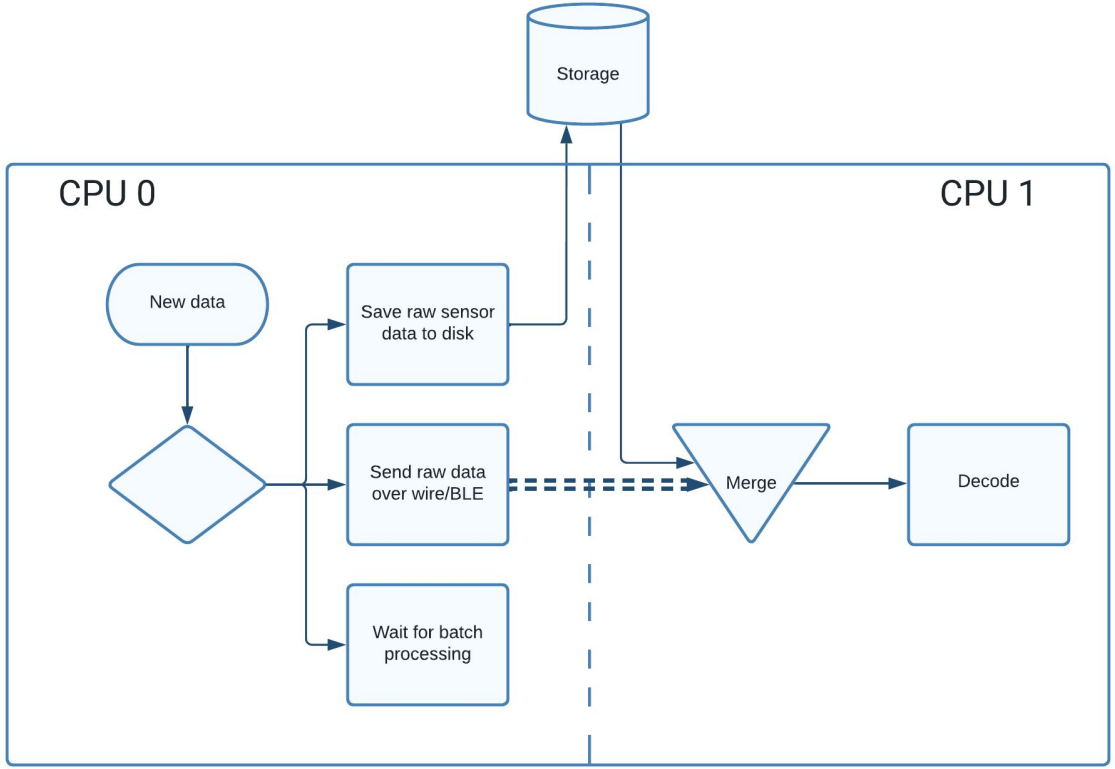
Future improvements

- Add more helpers to compliment `sensor_processing_with_callback()`
- Provide common API tests to verify decoders follow guidelines

Why do we need a decoder?

- When we get the data, it's stored in the RTIO mempool
- Allows batch processing of many samples
- We can get the decoder statically without an associated `struct device *`

This means...


Storage
CPU 0
CPU 1
New data
Save raw sensor data to disk
Send raw data over wire/BLE
Wait for batch processing
Merge
Decode

Getting the decoder at runtime

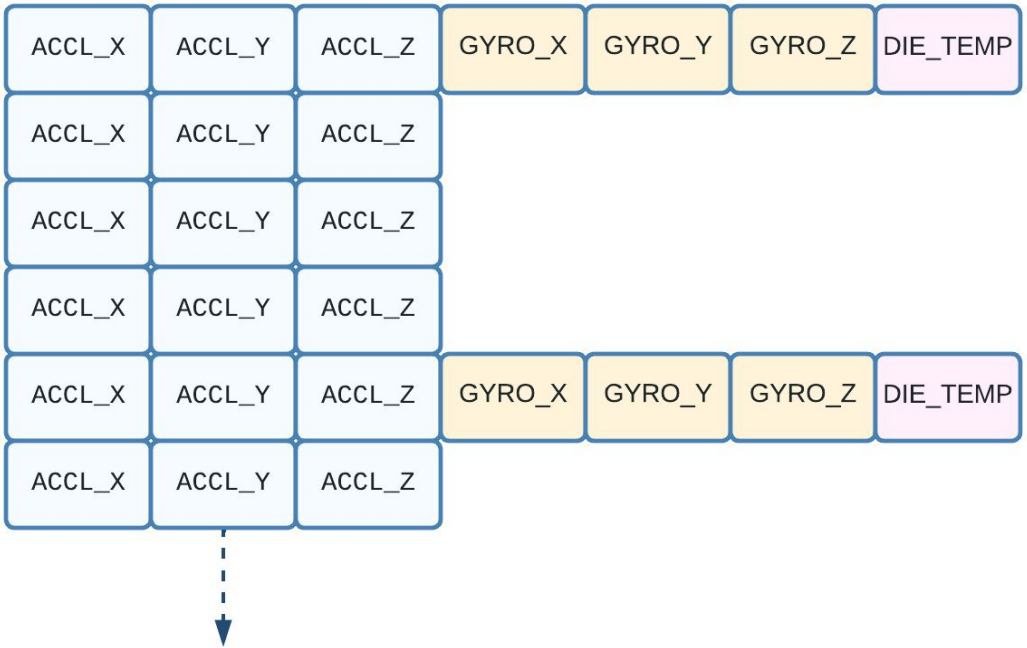
```
static void my_callback(int result, const uint8_t *buf,
                        uint32_t buf_len, void *userdata) {
  const struct device *dev = userdata;
  const struct sensor_decoder_api *decoder;

  int rc = sensor_get_decoder(dev, &decoder);
  if (rc != 0) {
    LOG_ERR("Failed to get decoder for '%s'", dev->name);
    return;
  }
  ...
}
```

Getting the timestamp of the sample

```
static void my_callback(int result, const uint8_t *buf,
                        uint32_t buf_len, void *userdata) {
  ...
  uint64_t timestamp_ns;
  rc = decoder->get_timestamp(buf, &timestamp_ns);
  if (rc != 0) {
    LOG_ERR("Failed to get timestamp");
    return;
  }
  LOG_INF("Sample fetched at %" PRIu64 "ns", timestamp_ns);
  ...
}
```

Data visualized



Decoding the samples: frames

- Frames are single snapshots in time
- All samples in the same frame were collected together

Example:

- Sensor is sampling at 100Hz
- Samples are collected over 20ms (2 frames)

For all one-shot reads, only 1 frame will ever be present.

Decoder iterators

- The `decode()` function takes 2 arguments:
  - `sensor_frame_iterator_t *fit`
  - `sensor_channel_iterator_t *cit`
- Both variables should be initialized using `= {0}` before decoding starts
- Return values are:
  - < 0: Error
  - 0: Nothing left to decode
  - > 0: The number of channels that were decoded

Telling when a frame ended

```
sensor_frame_iterator_t fit = {0}, fit_prev = {0};
sensor_channel_iterator_t cit = {0};
enum sensor_channel channel;
q31_t value;

// Read the samples 1 at a time
while (decoder->decode(buf, &fit, &cit,
                       &channel, &value, 1) > 0) {
  if (fit != fit_prev) {
    // New frame started
  }
  fit_prev = fit;
}
```

Understanding the q31_t data

- Provides a fixed point fractional value in the range of [-1, 1]
- Uses a shift to extend the range. Examples:
  - shift = 1, range = [-2, 2]
  - shift = -1, range = [-0.5, 0.5]
- Shift values provided by `decoder->get_shift()` are always the same when:
  - The samples are in the same buffer
  - The samples are of different axes of the same type (accel x/y/z)

Sample calculation using zDSP (accel magnitude)

```
q31_t xyz[3];
enum sensor_channel channels[3];
int rc = decoder->decode(buf, &fit, &cit,
                         xyz, channels, 3);

__ASSERT_NO_MSG(rc == 3);

int8_t shift;
decoder->get_shift(buf, channels[0], &shift);

// x=x^2, y=y^2, z=z^2
zdsp_mult_q31(xyz, xyz, xyz, 3);

// Saturating sum
int64_t sum = (int64_t)xyz[0] + (int64_t)xyz[1] + (int64_t)xyz[2];
sum = CLAMP(sum, INT32_MIN, INT32_MAX);

q31_t magn;
zdsp_sqrt_q31((q31_t)sum, &magn);
LOG_INF("Acceleration is %" PRIq(6) "m/s^2", PRIq_arg(magn, 6, shift));
```

Zephyr® Project
Developer Summit

Streaming data

What are streams?

- Anything that is interrupt driven (replacement for triggers)
  - steps, significant motion, tap events, etc.
- Some may include data
  - FIFO watermark

Setting up a stream reader

```
SENSOR_DT_STREAM_IODEV(
    my_stream,              // IODev name
    DT_CHOSEN(lid_accel), // Sensor node to stream
    SENSOR_STREAM_PREP(
        SENSOR_TRIG_FIFO_WATERMARK,
        SENSOR_STREAM_DATA_INCLUDE
    ),
    SENSOR_STREAM_PREP(...) // One or more streams
);
```

Setting up a stream reader

```
SENSOR_DT_STREAM_IODEV(
    my_stream,              // IODev name
    DT_CHOSEN(lid_accel), // Sensor node to stream
    SENSOR_STREAM_PREP(
        SENSOR_TRIG_FIFO_WATERMARK,
        SENSOR_STREAM_DATA_INCLUDE
    ),
    SENSOR_STREAM_PREP(...)
    // One or more streams
);
```

arg0: enum sensor_trigger_type
arg1: one of
- SENSOR_STREAM_DATA_INCLUDE
- SENSOR_STREAM_DATA_DROP
- SENSOR_STREAM_DATA_NOP

SENSOR_STREAM_DATA_INCLUDE

- Report the trigger fired
- Include any associated data

SENSOR_STREAM_DATA_DROP

- Report the trigger fired
- Flush and discard any associated data

SENSOR_STREAM_DATA_NOP

- Report the trigger fired
- Do nothing with the associated data

Example: using the FIFO full trigger. I want to know that my FIFO filled & read the data

SENSOR_STREAM_DATA_INCLUDE

- Report the trigger fired
- Include any associated data

SENSOR_STREAM_DATA_DROP

- Report the trigger fired
- Flush and discard any associated data

SENSOR_STREAM_DATA_NOP

- Report the trigger fired
- Do nothing with the associated data

Example: set this trigger mode for FIFO full to get a fresh start if you weren't able to process data fast enough.

SENSOR_STREAM_DATA_INCLUDE

- Report the trigger fired
- Include any associated data

SENSOR_STREAM_DATA_DROP

- Report the trigger fired
- Flush and discard any associated data

SENSOR_STREAM_DATA_NOP

- Report the trigger fired
- Do nothing with the associated data

Example: use this for a step detection but we don't care about the step count.

Starting the stream

```
struct rtio_sqe *handle;
int rc = sensor_stream(
    &my_stream,
    &sensor_read_rtio,
    /*userdata=*/ my_reader.sensor,
    &handle
);

if (rc != 0) {
  LOG_ERR("Failed to initiate stream (%d)", rc);
}
```

Stopping the stream

```
struct rtio_sqe *handle;
int rc = sensor_stream(
    &my_stream,
    &sensor_read_rtio,
    /*userdata=*/ my_reader.sensor,
    &handle
);

if (rc != 0) {
  LOG_ERR("Failed to initiate stream (%d)", rc);
}

rtio_sqe_cancel(handle);
```

Decoding triggers

- Triggers are a part of the data header

```
enum sensor_trigger_type triggers[5];
int offset = 0;
int num_triggers;

do {
  num_triggers = decoder->get_triggers(buffer, triggers, offset, /*max_count*/5);

  if (num_triggers <= 0) {
    LOG_ERR("Failed to read triggers");
    break;
  }

  offset += num_triggers;
  for (int i = 0; i < num_triggers; ++i) {
    LOG_INF("Trigger %d detected", triggers[i]);
  }
} while (num_triggers <= 5);
```

Summary

- Using the RTIO mempool allows for memory granularity control
- Removing interrupt processing from sensors
  - Removes per sensor thread (wastes memory)
  - Removes need to use system work queue (hard to configure and timing isn’t reliable)
- One-shot and streaming data paths are the same
- Finer control over what happens when a trigger is detected

Zephyr® Project
Developer Summit

Questions?