Zephyr® Project

CROXEL

RTIO for Real-Time Zephyr Apps

Luis Ubieda
Lead Firmware Engineer

@ubieda luisf@croxel.com

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA

Agenda

1. Introduction
2. Basics of RTIO
3. The *Async* vs *Sync* Paradigms
4. In-tree example of RTIO
5. RTIO and Real-Time
6. Demo - RTIO Performance Benchmarks
7. The Future of RTIO
8. Q&A

Introduction

Basics of RTIO

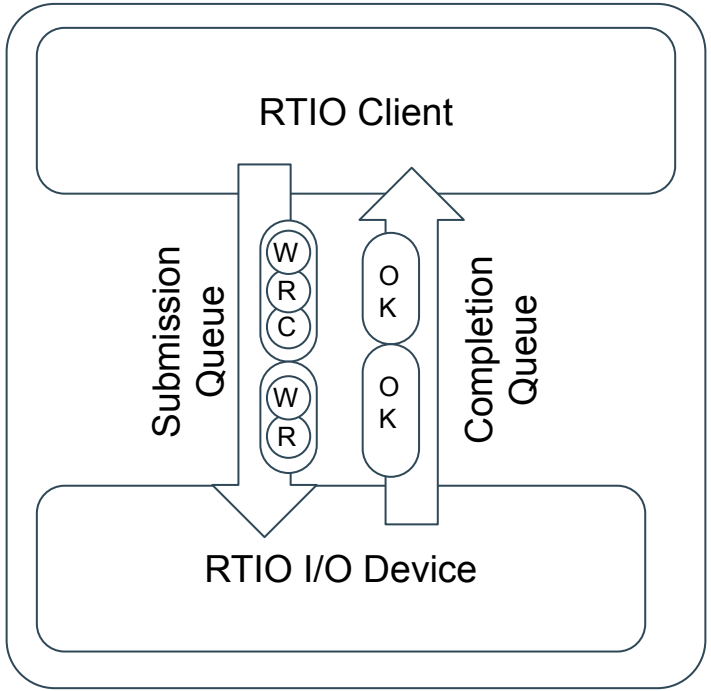
Basics of RTIO

Elements of an RTIO

- RTIO Client
- RTIO I/O Device
- RTIO Executor
- Optional: a Memory Pool.
- A Submission Queue
- A Completion Queue

Types of Operations:

- Write
- Read
- Callback
- Delay



Examples of I/O Devices:

- Sensors
- I/O Buses (e.g: SPI, I2C, I3C, etc)

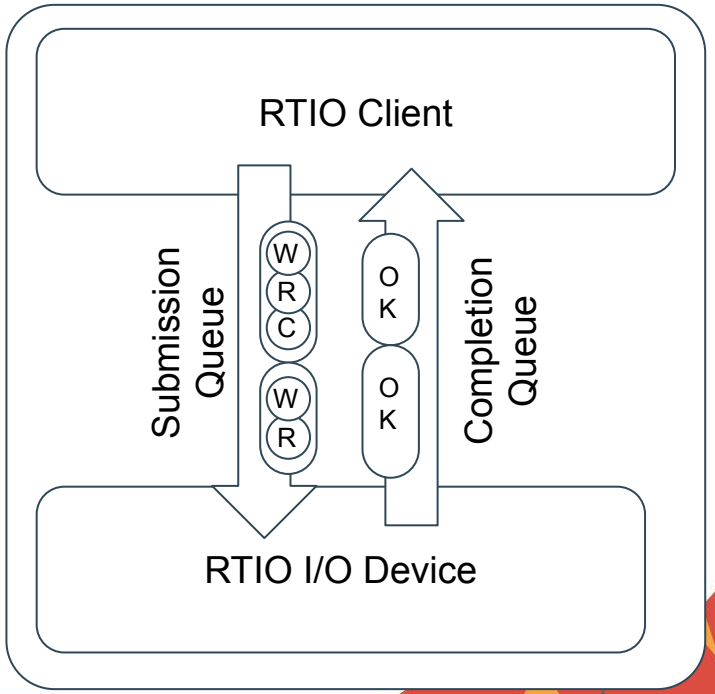
Example Applications:

- Control Systems (e.g: Robotics)
- High-Bandwidth Data-Acquisition Systems

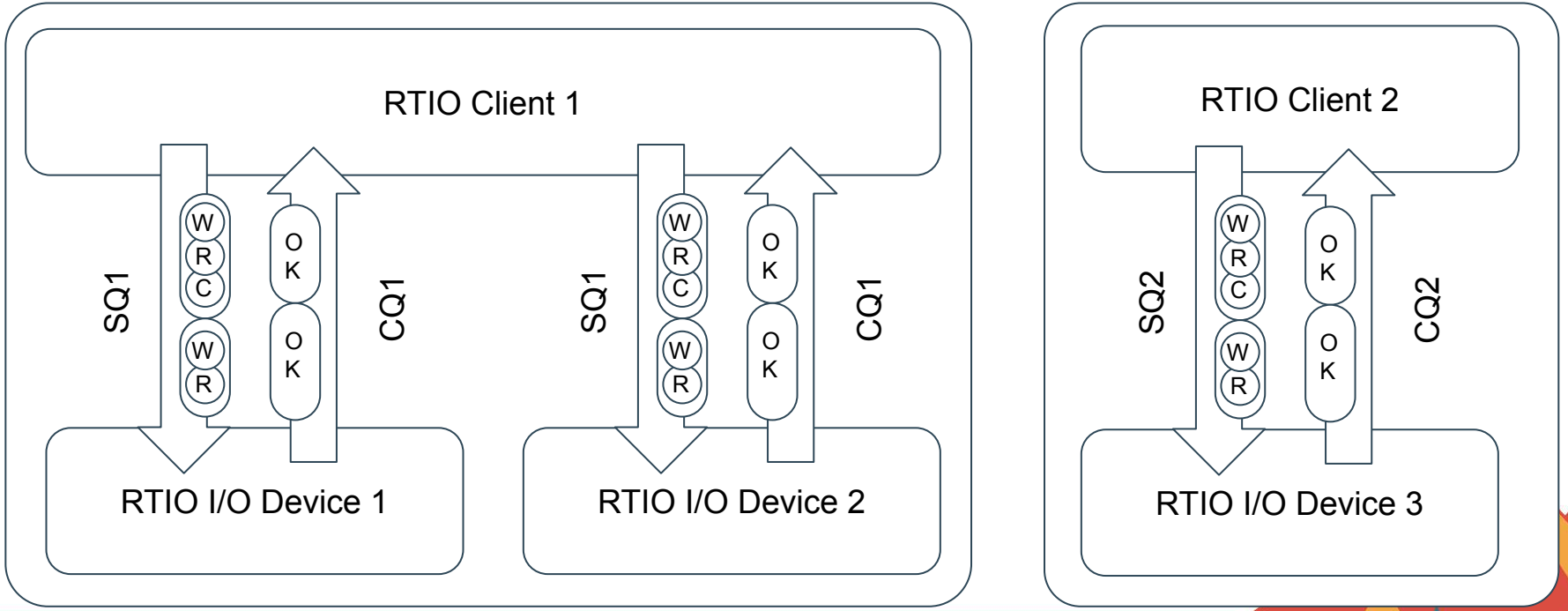
Basics of RTIO

Design Goals of RTIO

- Minimize thread context switching
- Minimize data copying
- Optimize batch gathering and processing of multiple data-streams
- Compatible with Userspace-based applications
- Side-effect: serves as a IODEV abstraction (e.g: Bus-driver)



Basics of RTIO

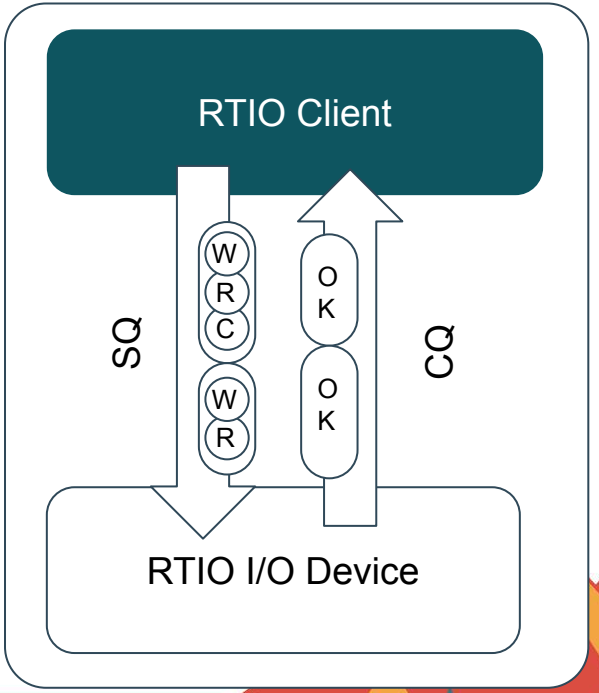


Basics of RTIO - Client

```
/* Instantiate an RTIO client and IODEV reading instance(s) */
RTIO_DEFINE(r_client, SQ_SZ, CQ_SZ);
RTIO_IODEV_DEFINE(iodev_1, &iodev_api, &iodev_read_cfg);
```

```
/* Submit a Write request to an IODEV */
struct rtio_sqe *sqe = rtio_sqe_acquire(&r_client);

rtio_sqe_prep_write(sqe, iodev, PRIO, NULL);

ret = rtio_submit(&r_client_1, WAIT_CT);
```

```
struct rtio_cqe *cqe = rtio_cqe_consume(&r_client);

if ((cqe != NULL) && (cqe->result < 0)) {
        /* Handle error */
}
rtio_cqe_release(&r_client, cqe);
```

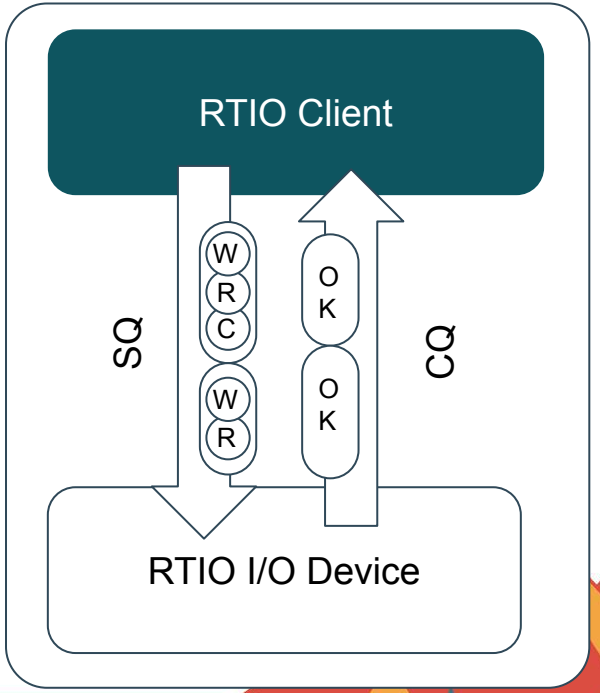


Basics of RTIO - Client

```
/* Instantiate an RTIO client and IODEV reading instance */
RTIO_DEFINE_WITH_MEMPOOL(r_client, SQ_SZ, CQ_SZ,
                         BLK_NUM, BLK_SZ, BLK_ALIGN);
RTIO_IODEV_DEFINE(iodev_1, &iodev_api, &iodev_read_cfg);
```

```
/* Submit a Write request to an IODEV */
struct rtio_sqe *sqe = rtio_sqe_acquire(&r_client);

rtio_sqe_prep_read_with_pool(sqe, iodev, PRIO, NULL);

ret = rtio_submit(&r_client_1, WAIT_CT);
```

```
struct rtio_cqe *cqe = rtio_cqe_consume(&r_client);

if ((cqe != NULL) && (cqe->result < 0)) {
       /* Handle error */
}
rtio_cqe_get_mempool_buffer(ctx, cqe, &buf, &buf_len);
rtio_cqe_release(&r_client, cqe);

/* Process data */

rtio_release_buffer(&r_client, buf, buf_len);
```

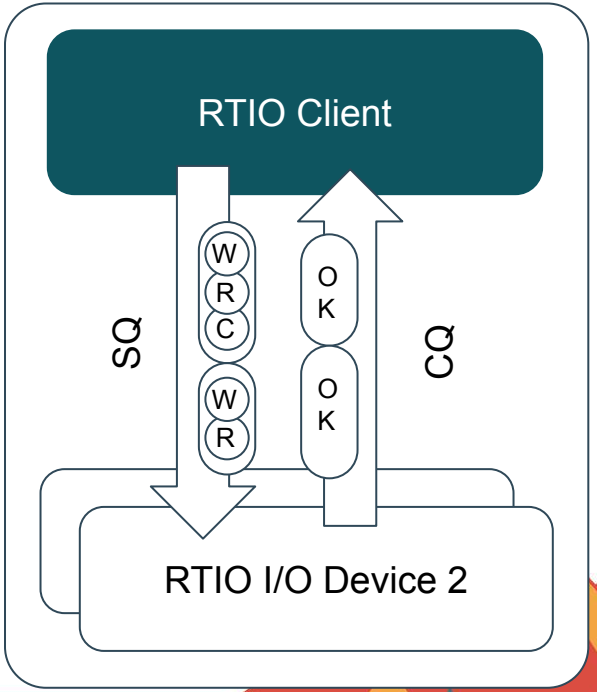


Basics of RTIO - Client

```c
/* Instantiate an RTIO client and IODEV reading instance(s) */
RTIO_DEFINE(r_client, SQ_SZ, CQ_SZ);
RTIO_IODEV_DEFINE(iodev_1, &iodev_api, &iodev_read_cfg);
RTIO_IODEV_DEFINE(iodev_2, &iodev_api, &iodev_read_cfg);
```

```c
/* Submit a Write request to an IODEV */
struct rtio_sqe *sqe_wr1 = rtio_sqe_acquire(&r_client);
struct rtio_sqe *sqe_wr2 = rtio_sqe_acquire(&r_client);

rtio_sqe_prep_write(sqe_wr1, iodev_1, PRIO, NULL);
rtio_sqe_prep_write(sqe_wr2, iodev_2, PRIO, NULL);

ret = rtio_submit(&r_client_1, WAIT_CT);
```

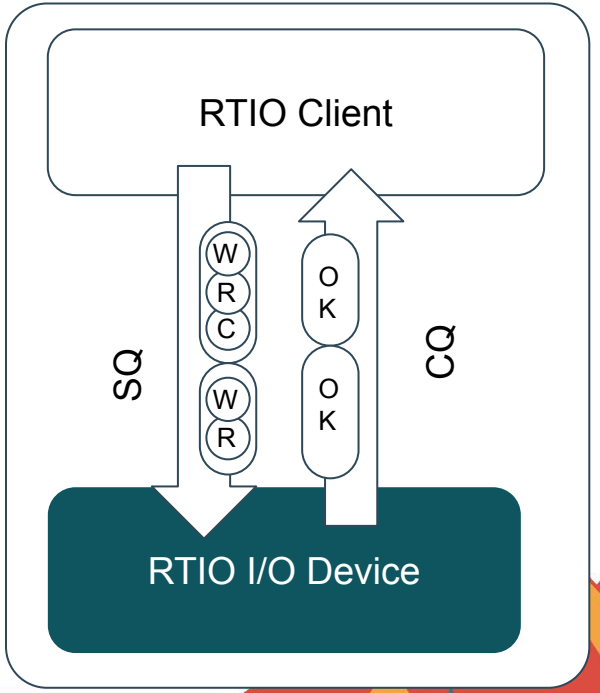
```c
struct rtio_cqe *cqe;

do {
    cqe = rtio_cqe_consume(&r_client);
    if ((cqe != NULL) && (cqe->result < 0)) {
        /* Handle error*/
    }
    rtio_cqe_release(&r_client, cqe);
} while (cqe != NULL);
```



Basics of RTIO - I/O Device

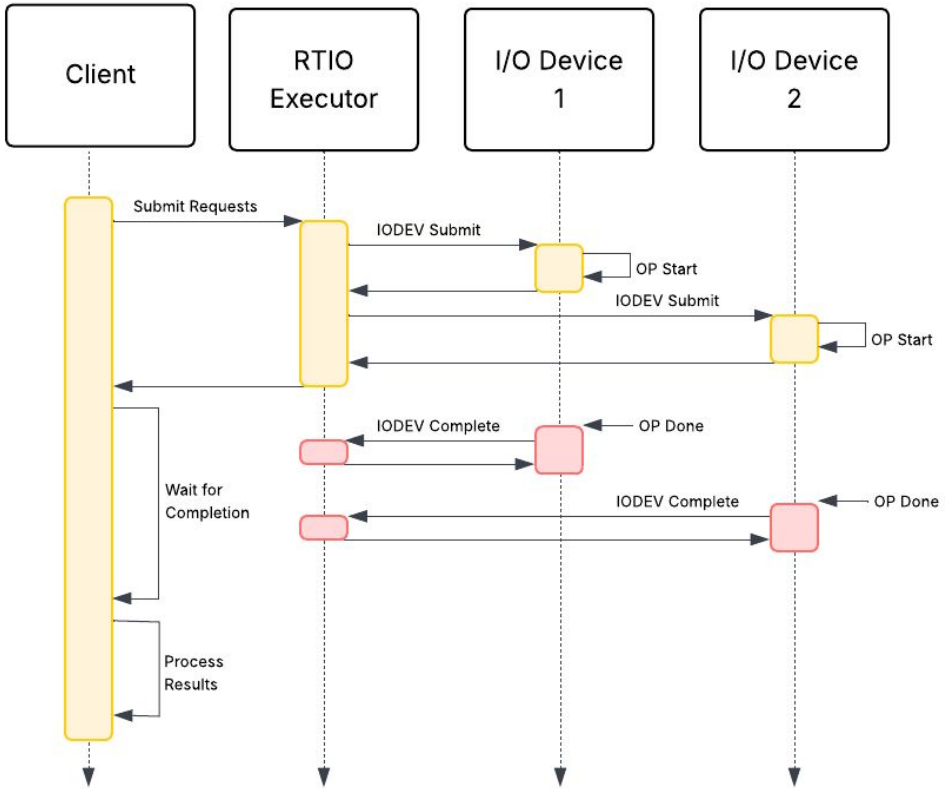
```c
static void iodev_op_result(struct rtio_iodev_sqe *iodev_sqe, int result)
{
   if (result == 0) {
       rtio_iodev_sqe_ok(iodev_sqe, 0);
   } else {
       rtio_iodev_sqe_err(iodev_sqe, result);
   }
}
/* Handle Submission items (Must be non-blocking) */
static void iodev_submit(struct rtio_iodev_sqe *iodev_sqe)
{
   /* Identify OP Requested and kick-off process */
}
```

```c
struct rtio_iodev_api iodev_1_api = {
    .submit = iodev_submit,
}
```

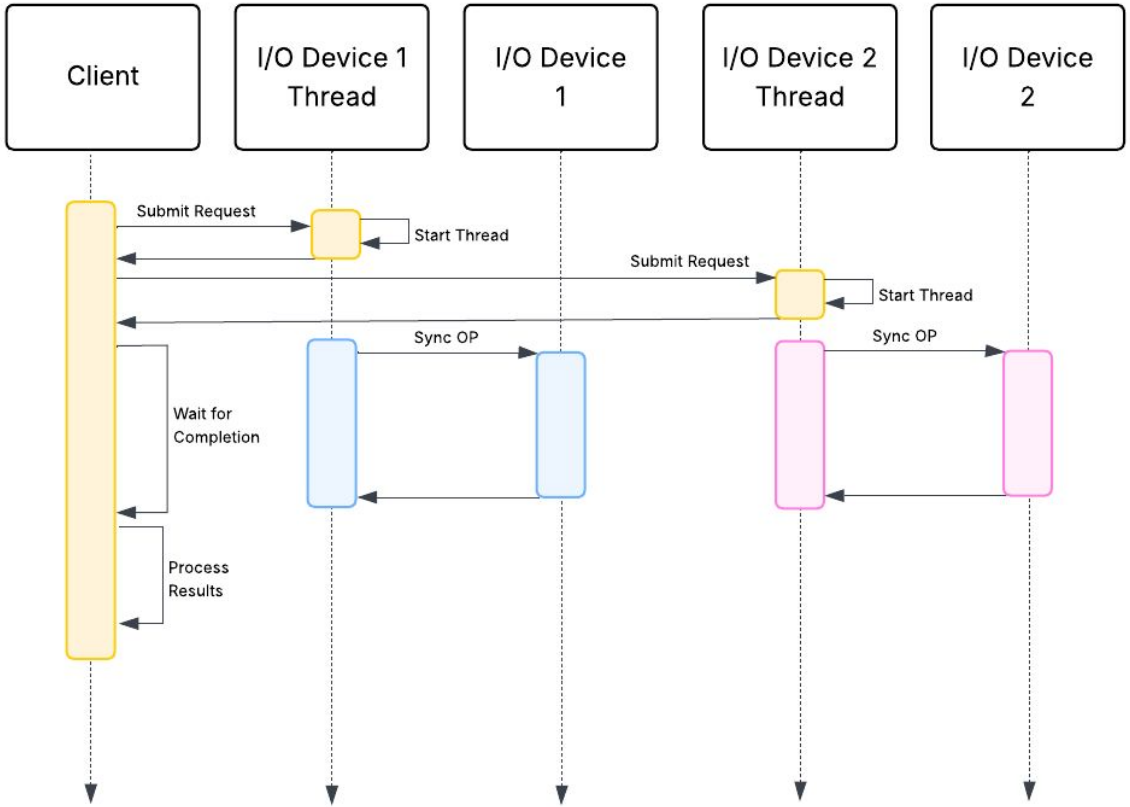


The *Async* vs *Sync* Paradigms

RTIO: The *Async* Paradigm



The *Sync* Paradigm



RTIO in the Zephyr Tree

In-tree Example - Sensors

```
#define SENSOR_DT_READ_IODEV(name, dt_node, ...)
#define SENSOR_DT_STREAM_IODEV(name, dt_node, ...)


/* Synchronous/Blocking sensor read */

static inline int sensor_read(struct rtio_iodev *iodev, struct rtio *ctx,
                              uint8_t *buf, size_t buf_len);


/* Asynchronous Sensor Read */

static inline int sensor_read_async_mempool(struct rtio_iodev *iodev,
                                            struct rtio *ctx, void *userdata);

/* Process Read requests */

void sensor_processing_with_callback(struct rtio *ctx,
                                     sensor_processing_callback_t cb);


typedef void (*sensor_processing_callback_t)(int result,
                                             uint8_t *buf, uint32_t buf_len,
                                             void *userdata);


/* Start a stream of data (events) */

static inline int sensor_stream(struct rtio_iodev *iodev, struct rtio *ctx,
                                void *userdata, struct rtio_sqe **handle);
```

In-tree Example - Sensors

```c
#define SENSOR_DT_READ_IODEV(name, dt_node, ...)
#define SENSOR_DT_STREAM_IODEV(name, dt_node, ...)

/* Synchronous/Blocking sensor read */

static inline int sensor_read(struct rtio_iodev *iodev, struct rtio *ctx,
                              uint8_t *buf, size_t buf_len);

/* Asynchronous Sensor Read */

static inline int sensor_read_async_mempool(struct rtio_iodev *iodev,
                                            struct rtio *ctx, void *userdata);

/* Process Read requests */

void sensor_processing_with_callback(struct rtio *ctx,
                                     sensor_processing_callback_t cb);

typedef void (*sensor_processing_callback_t)(int result,
                                             uint8_t *buf, uint32_t buf_len,
                                             void *userdata);

/* Start a stream of data (events) */

static inline int sensor_stream(struct rtio_iodev *iodev, struct rtio *ctx,
                                void *userdata, struct rtio_sqe **handle);
```

```c
static inline int sensor_read(struct rtio_iodev *iodev, struct rtio *ctx, uint8_t *buf,
                              size_t buf_len)
{
        if (IS_ENABLED(CONFIG_USERSPACE)) {
                struct rtio_sqe sqe;

                rtio_sqe_prep_read(&sqe, iodev, RTIO_PRIO_NORM, buf, buf_len, buf);
                rtio_sqe_copy_in(ctx, &sqe, 1);
        } else {
                struct rtio_sqe *sqe = rtio_sqe_acquire(ctx);

                if (sqe == NULL) {
                        return -ENOMEM;
                }
                rtio_sqe_prep_read(sqe, iodev, RTIO_PRIO_NORM, buf, buf_len, buf);
        }
        rtio_submit(ctx, 0);

        struct rtio_cqe *cqe = rtio_cqe_consume_block(ctx);
        int res = cqe->result;

        __ASSERT(cqe->userdata == buf,
                 "consumed non-matching completion for sensor read into buffer %p\n", buf);

        rtio_cqe_release(ctx, cqe);

        return res;
}
```

In-tree Example - Sensors

```c
#define SENSOR_DT_READ_IODEV(name, dt_node, ...)
#define SENSOR_DT_STREAM_IODEV(name, dt_node, ...)

/* Synchronous/Blocking sensor read */

static inline int sensor_read(struct rtio_iodev *iodev, struct rtio *ctx,
                              uint8_t *buf, size_t buf_len);

/* Asynchronous Sensor Read */

static inline int sensor_read_async_mempool(struct rtio_iodev *iodev,
                                            struct rtio *ctx, void *userdata);

/* Process Read requests */

void sensor_processing_with_callback(struct rtio *ctx,
                                     sensor_processing_callback_t cb);

typedef void (*sensor_processing_callback_t)(int result,
                                             uint8_t *buf, uint32_t buf_len,
                                             void *userdata);

/* Start a stream of data (events) */

static inline int sensor_stream(struct rtio_iodev *iodev, struct rtio *ctx,
                                void *userdata, struct rtio_sqe **handle);
```

```c
static inline int sensor_read_async_mempool(struct rtio_iodev *iodev, struct rtio *ctx,
                                            void *userdata)
{
        if (IS_ENABLED(CONFIG_USERSPACE)) {
                struct rtio_sqe sqe;

                rtio_sqe_prep_read_with_pool(&sqe, iodev, RTIO_PRIO_NORM, userdata);
                rtio_sqe_copy_in(ctx, &sqe, 1);
        } else {
                struct rtio_sqe *sqe = rtio_sqe_acquire(ctx);

                if (sqe == NULL) {
                        return -ENOMEM;
                }
                rtio_sqe_prep_read_with_pool(sqe, iodev, RTIO_PRIO_NORM, userdata);
        }
        rtio_submit(ctx, 0);
        return 0;
}
```

In-tree Example - Sensors

```c
#define SENSOR_DT_READ_IODEV(name, dt_node, ...)
#define SENSOR_DT_STREAM_IODEV(name, dt_node, ...)

/* Synchronous/Blocking sensor read */

static inline int sensor_read(struct rtio_iodev *iodev, struct rtio *ctx,
                              uint8_t *buf, size_t buf_len);

/* Asynchronous Sensor Read */

static inline int sensor_read_async_mempool(struct rtio_iodev *iodev,
                                            struct rtio *ctx, void *userdata);

/* Process Read requests */

void sensor_processing_with_callback(struct rtio *ctx,
                                     sensor_processing_callback_t cb);

typedef void (*sensor_processing_callback_t)(int result,
                                             uint8_t *buf, uint32_t buf_len,
                                             void *userdata);

/* Start a stream of data (events) */

static inline int sensor_stream(struct rtio_iodev *iodev, struct rtio *ctx,
                                void *userdata, struct rtio_sqe **handle);
```

```c
void sensor_processing_with_callback(struct rtio *ctx, sensor_processing_callback_t cb)
{
        void *userdata = NULL;
        uint8_t *buf = NULL;
        uint32_t buf_len = 0;
        int rc;

        /* Wait for a CQE */
        struct rtio_cqe *cqe = rtio_cqe_consume_block(ctx);

        /* Cache the data from the CQE */
        rc = cqe->result;
        userdata = cqe->userdata;
        rtio_cqe_get_mempool_buffer(ctx, cqe, &buf, &buf_len);

        /* Release the CQE */
        rtio_cqe_release(ctx, cqe);

        /* Call the callback */
        cb(rc, buf, buf_len, userdata);

        /* Release the memory */
        rtio_release_buffer(ctx, buf, buf_len);
}
```

In-tree Example - Sensors

```c
#define SENSOR_DT_READ_IODEV(name, dt_node, ...)
#define SENSOR_DT_STREAM_IODEV(name, dt_node, ...)

/* Synchronous/Blocking sensor read */

static inline int sensor_read(struct rtio_iodev *iodev, struct rtio *ctx,
                              uint8_t *buf, size_t buf_len);

/* Asynchronous Sensor Read */

static inline int sensor_read_async_mempool(struct rtio_iodev *iodev,
                                            struct rtio *ctx, void *userdata);

/* Process Read requests */

void sensor_processing_with_callback(struct rtio *ctx,
                                     sensor_processing_callback_t cb);

typedef void (*sensor_processing_callback_t)(int result,
                                             uint8_t *buf, uint32_t buf_len,
                                             void *userdata);

/* Start a stream of data (events) */

static inline int sensor_stream(struct rtio_iodev *iodev, struct rtio *ctx,
                                void *userdata, struct rtio_sqe **handle);
```

```c
static inline int sensor_stream(struct rtio_iodev *iodev, struct rtio *ctx, void *userdata,
                                struct rtio_sqe **handle)
{
        if (IS_ENABLED(CONFIG_USERSPACE)) {
                struct rtio_sqe sqe;

                rtio_sqe_prep_read_multishot(&sqe, iodev, RTIO_PRIO_NORM, userdata);
                rtio_sqe_copy_in_get_handles(ctx, &sqe, handle, 1);
        } else {
                struct rtio_sqe *sqe = rtio_sqe_acquire(ctx);

                if (sqe == NULL) {
                        return -ENOMEM;
                }
                if (handle != NULL) {
                        *handle = sqe;
                }
                rtio_sqe_prep_read_multishot(sqe, iodev, RTIO_PRIO_NORM, userdata);
        }
        rtio_submit(ctx, 0);
        return 0;
}
```

RTIO and Real Time

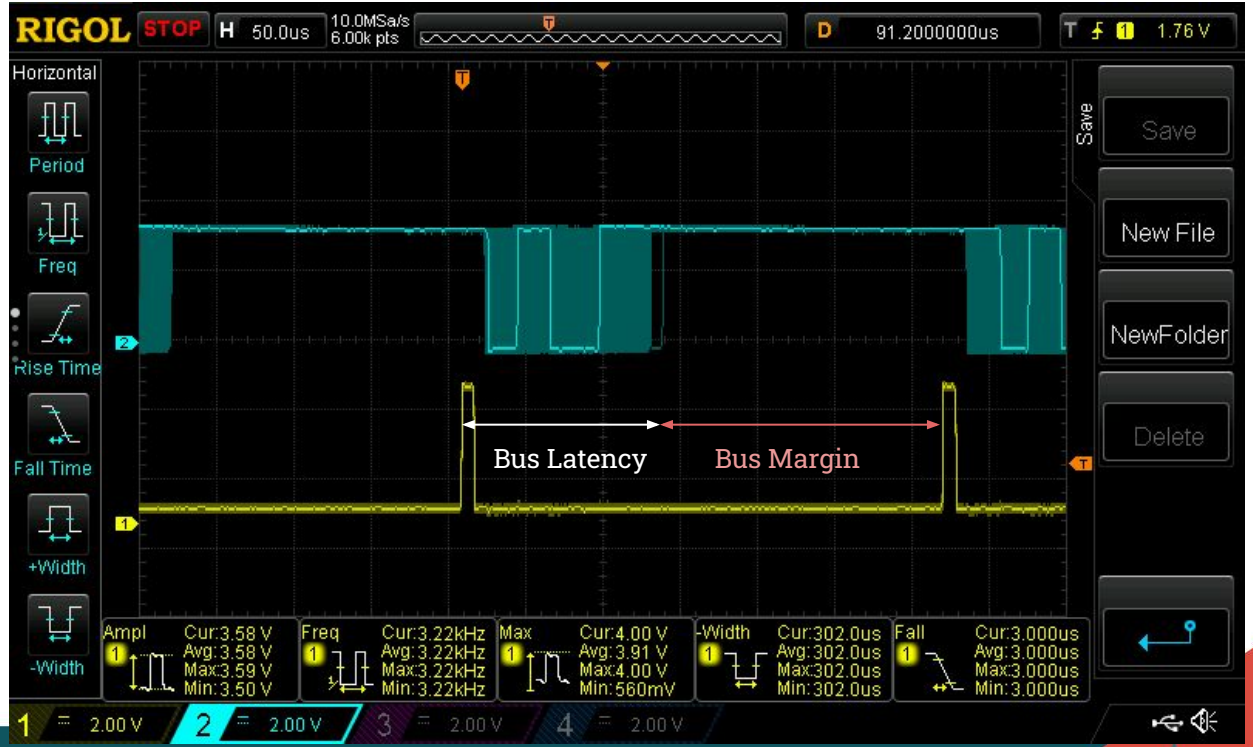
Real Time in I/O

Real-Time Constraints

Perform actions with guaranteed minimum constraints

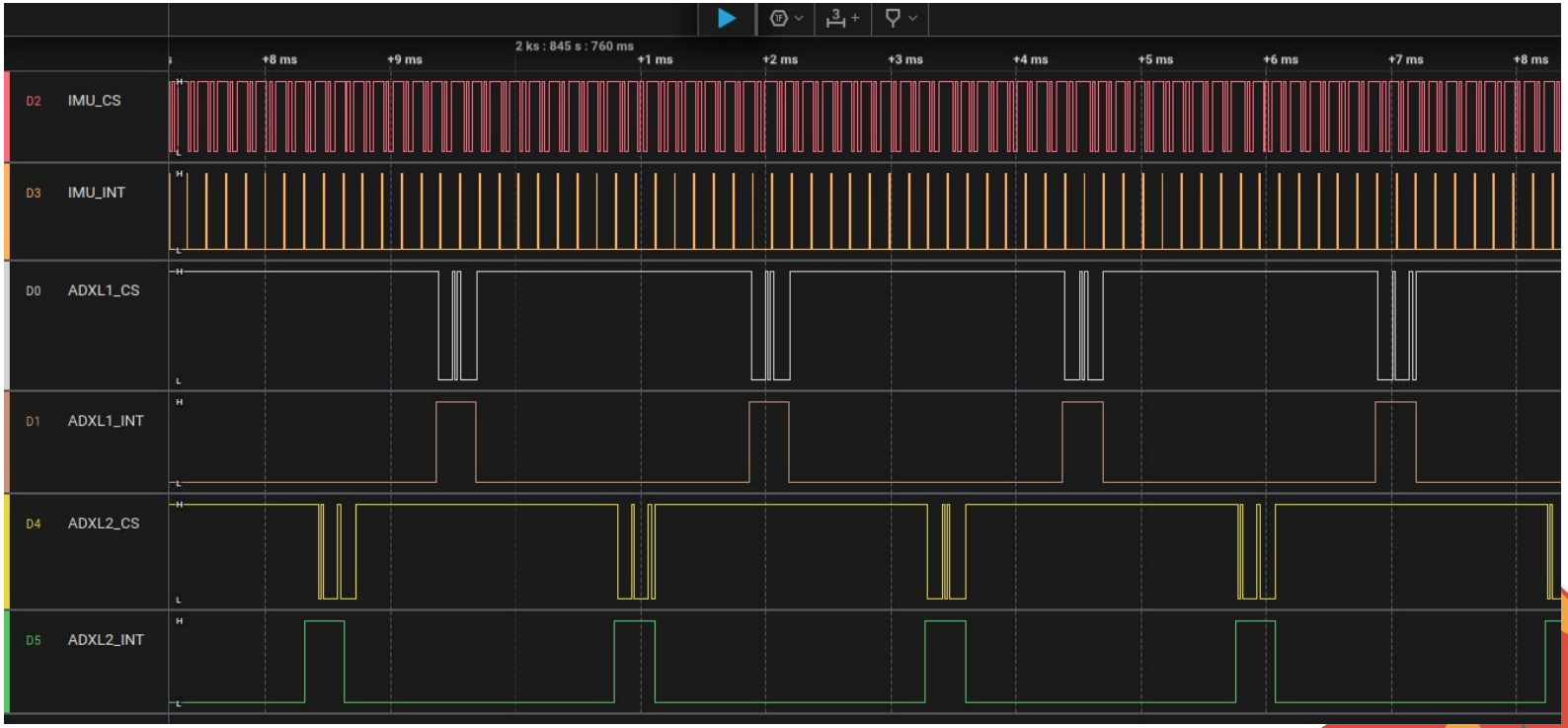
- Maximum Allowed Latency
  - Reaction Time
  - Bus Latency
  - Jitter
- CPU Cycles to perform all the required actions
  - Data Collection
  - Data Processing/Transport
- Conditions validated under worst case scenario(s)

Real Time in I/O



Real Time in RTIO

Real Time in RTIO

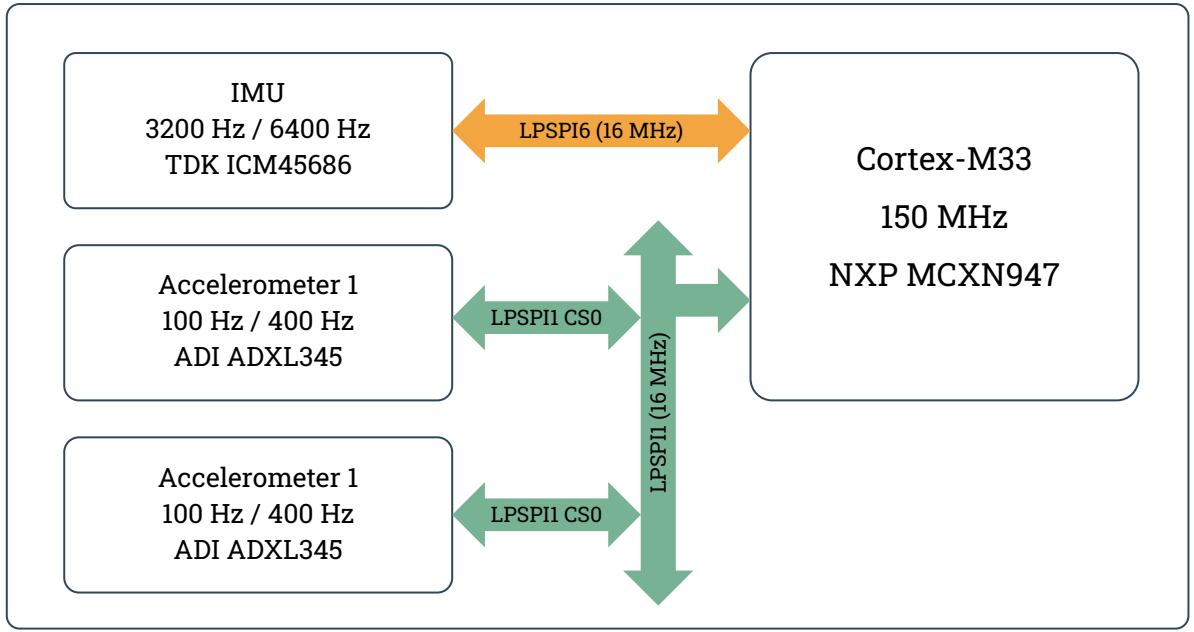


RTIO in Real Time

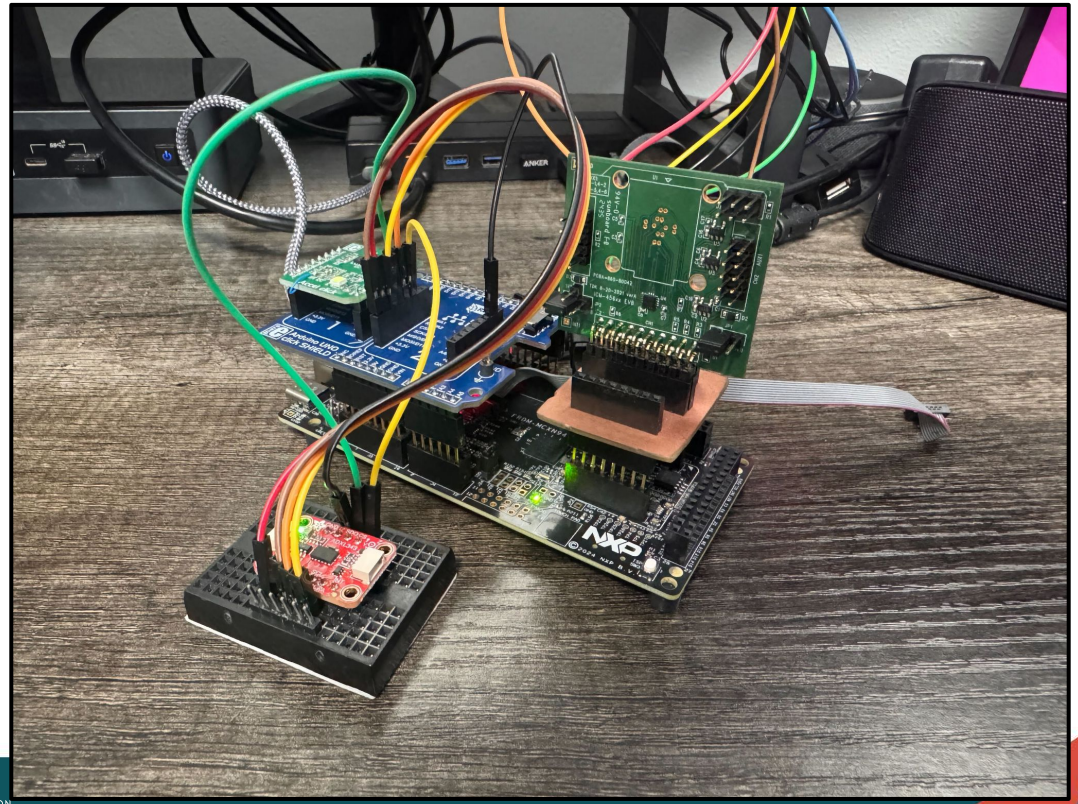
- **Minimize Overhead**
  - Memory Copying
  - Data conversions
  - Context Swapping
- **Designed for Real Time**
  - Take advantage of Hardware features (e.g: FIFO, DMA)
  - Ease to set up simultaneous streams of data (Async paradigm)
  - Guarantees decoupling of data-acquisition and processing
- **Optimized Results**
  - Latency
  - Jitter
  - CPU Load

RTIO Performance Benchmarks

Setup - Device Under Test



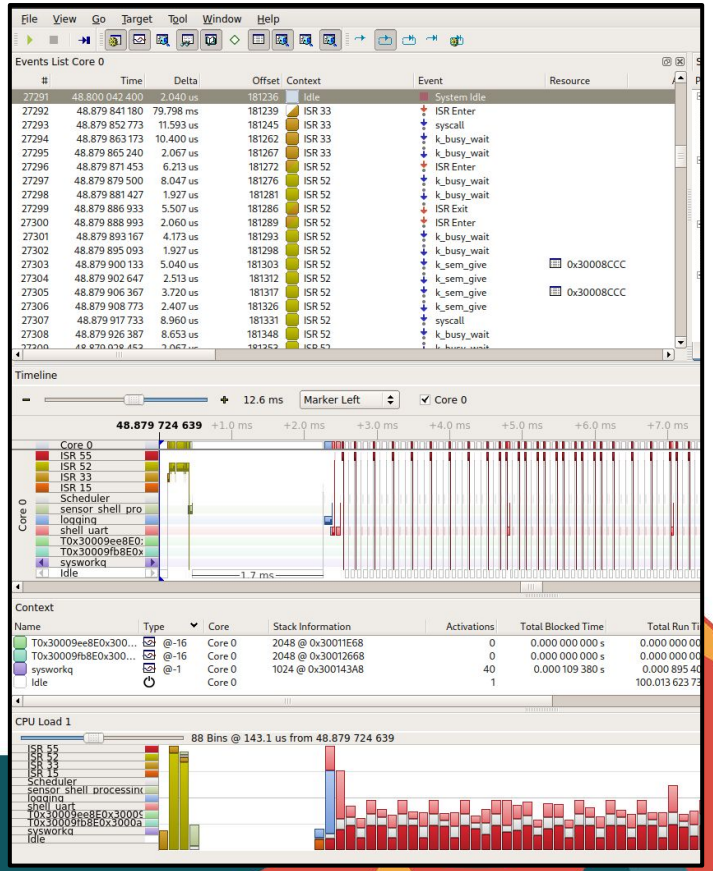
Setup - Device Under Test



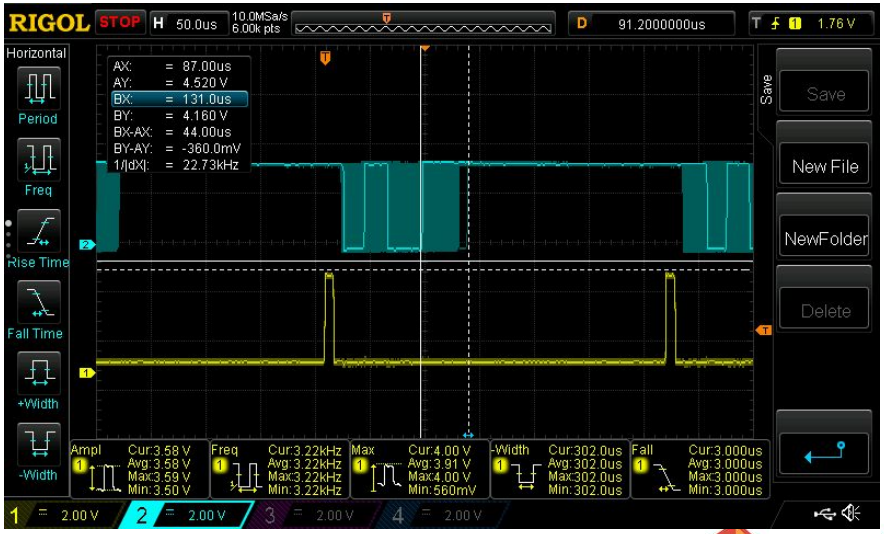
Setup - Measuring CPU Load

```
# Overlay Configuration
CONFIG_CPU_LOAD=y
CONFIG_CPU_LOAD_LOG_PERIODICALLY=5000
```

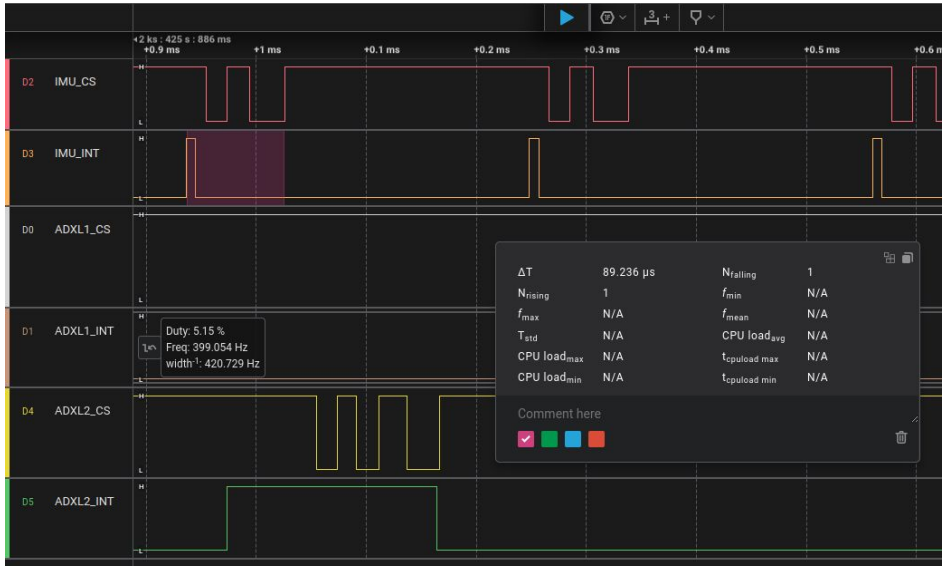
```
[00:16:10.015,000] <inf> cpu_load: Load:47.003%
[00:16:15.015,000] <inf> cpu_load: Load:47.003%
[00:16:20.015,000] <inf> cpu_load: Load:47.003%
[00:16:25.015,000] <inf> cpu_load: Load:47.003%
[00:16:30.015,000] <inf> cpu_load: Load:47.002%
[00:16:35.015,000] <inf> cpu_load: Load:47.003%
[00:16:40.015,000] <inf> cpu_load: Load:47.003%
[00:16:45.015,000] <inf> cpu_load: Load:47.003%
[00:16:50.015,000] <inf> cpu_load: Load:47.003%
[00:16:55.015,000] <inf> cpu_load: Load:47.003%
[00:17:00.015,000] <inf> cpu_load: Load:47.003%
[00:17:05.015,000] <inf> cpu_load: Load:47.003%
[00:17:10.016,000] <inf> cpu_load: Load:47.003%
[00:17:15.016,000] <inf> cpu_load: Load:47.003%
[00:17:20.016,000] <inf> cpu_load: Load:47.003%
uart:~$
  bypass              clear               date
  demo                device              devmem
  dynamic             help                history
  kernel              log                 log_test
  rem                 resize              retval
  section_cmd         sensor              shell
  shell_uart_release  stats               version
uart:~$
```

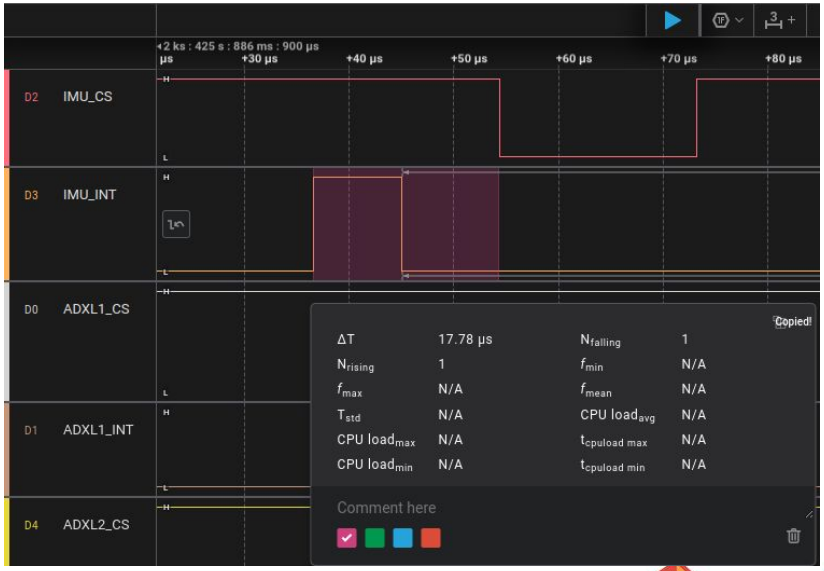


Setup - Measuring Jitter



Setup - Measuring Bus Latency





RTIO Performance Benchmarks

Settings		CPU Load (%)		Latency (us)		Jitter (us)	
		Thread-based	RTIO-based	Thread-based	RTIO-based	Thread-based	RTIO-based
IMU at 3200Hz ODR	No Simultaneous Stream	42	35	92	90.6	37.2	11.6
	Simult. 1 Accel at 100Hz ODR	43	36	93	89.7	92	32
	Simult. 2 Accels at 100Hz ODR	45	38	92	89.7	116	41.6
	Simult. 1 Accel at 400Hz ODR	48	41	93	91	175	44
	Simult. 2 Accels at 400Hz ODR	55	47	92	90	155	44
IMU at 6400Hz ODR	No Simultaneous Stream	83	72	90	92	63	12.8
	Simult. 1 Accel at 100Hz ODR	Missing events	72	Missing events	93	Missing events	34.4
	Simult. 2 Accels at 100Hz ODR	Missing events	74	Missing events	89	Missing events	44.4
	Simult. 1 Accel at 400Hz ODR	Missing events	76	Missing events	90	Missing events	37.2
	Simult. 2 Accels at 400Hz ODR	Missing events	80	Missing events	91	Missing events	48

The Future of RTIO

The Future of RTIO

1. Extend Existing Support

- Sensors (V2 Model)
- Bus Drivers

2. Expand to more Applications

- ADC (In-Flight PR)
- Video (in discussion)
- Audio (TBD)
- Peripherals Interconnection (TBD)

The Future of RTIO

Join and Engage with the Zephyr Community!

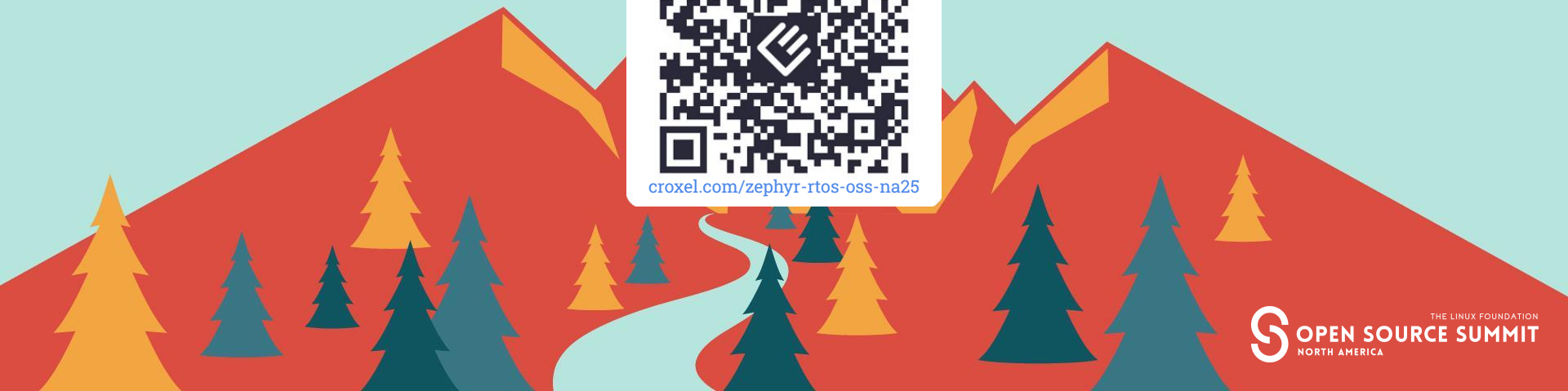
- Discord RTIO channel
- Sensors Working Group - Biweekly, Mondays @ 7AM PT

CROXEL

Zephyr® Project

Let's talk about Zephyr!

croxel.com/zephyr-rtos-oss-na25



THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA

Q&A