

David Noronha

Origin Assignment

davidnoronha@outlook.in

↘ means its a link you can click on.

I hope you can read my handwriting, Be very liberal w/ Initial Conditions. zooming in & out.

Hardware:

↘ Jetson AGX Orin (Ubuntu 22.04 LTS + ROS2 Humble)
↘ Liuxon Mid-360

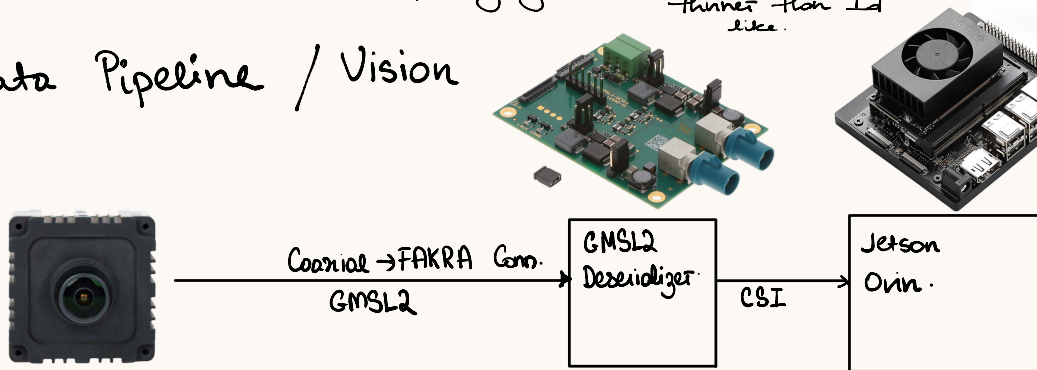
e- Con Systems STURDeCAM25

↘ Universal Robotics UR7e → 6 DOF Arm.
Ethernet → PTPx2 IEEE 1588

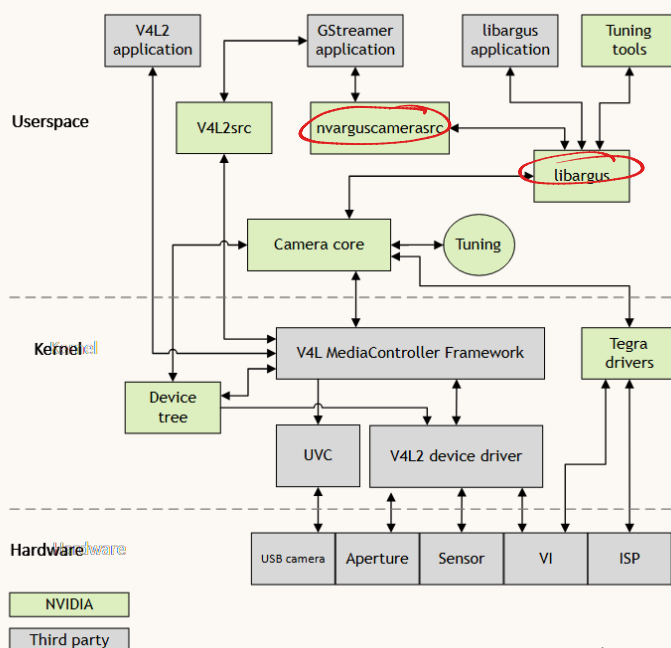
I've tried to link source docs as much as I can but sometimes I've forgotten when I'm in a research rabbit hole.

3) Whatever I've had trouble finding good docs are a little thinner than I'd like.

Data Pipeline / Vision



I've tried my best to follow the spirit of the doc



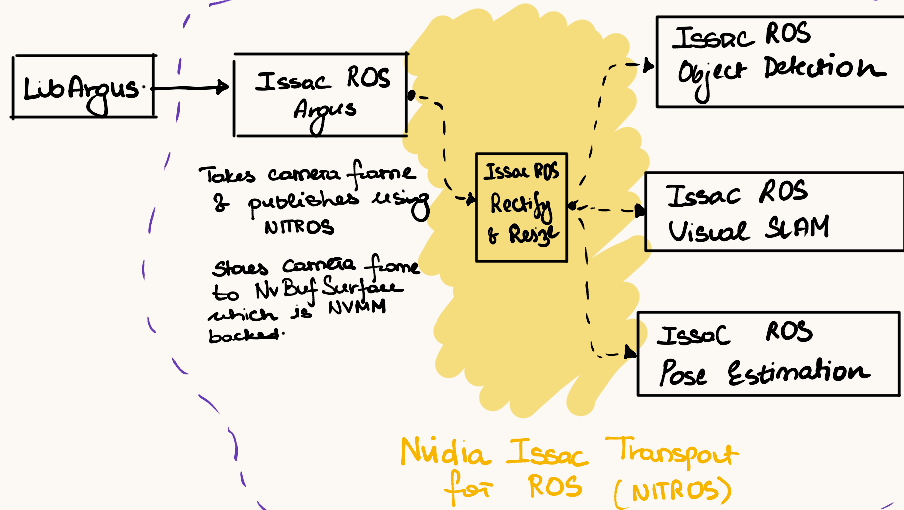
V4L2 Camera Support in Nvidia Jetson.
(from Tegra Linux docs)

Modern ISSAC ROS for the Jetson platform. Recommends Nvidia SIPL for the camera but doesn't seem to be stable / reliable for the Orin, using older version of ISSAC-ROS (v3.3)

Camera Supported Output:
UYVY pixel format

1280 × 720	120 f/s
1920 × 1080	65 f/s
1920 × 1200	60 f/s

* - From Camera Datasheet.



NITROS nodes that are sent to each other in a graph, they apply type negotiation & type adoption which removes memory copies.

Over ROS each node simply publishes a small header w/ the location to the message/data in CPU memory.

GPU side pointers / NVBufSurface internally backed by NVMM

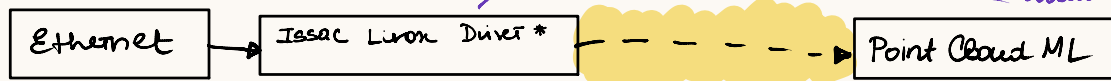
Composable Node Container (component_container_mt)

Also need to compose the nodes using 'component_container_mt' to allow intra process communication.

Since if you need to do interprocess communication, you need to use DDS/RMW which brings copies.

Data Pipeline/Point cloud

similar flow only using NITROS to enable zero copy transfer across our stack. → the camera & LIDAR flow would mostly run in parallel / together. I've kept them separate for clarity.



*There exists no Issac ROS Livox Driver so a custom one utilizing GXF & the base Livox SDK would have to be made.

We can either make a bridge node that consumes the PointCloud message & republishes into NITROS.

or modify the Livox Driver to work similarly to Issac ROS Nasa / Issac ROS Hesai which is a NITROS accelerated driver for the Hesai LIDAR.

But this only makes sense if our computation using the pointcloud is GPU accel. & can make use of the data being available in the GPU to reduce copies. If the processing is mostly CPU bound then ROS2 Intra Process communication (Sharing memory within process) should be sufficient.

Additional Details:

- If using ROS a 1080p raw image at 65fps is $1920 \times 1080 \times 3 \times 65 \text{ bytes/s}$ $\Rightarrow 404.352 \text{ Megabytes/s}$. This data is copied multiple times during serialization & deserialization. This eats up memory bandwidth.
- The TensorRT inference node can automatically use the Nitros Image Handles w/o any copies. Removing need for copy from Host to Device.
- Can run inference on the Jetson's DLA to offload processing from the GPU.
- In the LIDAR driver, could use Remote DMA if available to directly write the received sensor data to the Onix memory. (accessible by the GPU & GPU)
- Can test with using Cyclone DDS with shared memory to use latency in the pipeline, especially between 2 regular ROS nodes.

2) System Architecture & Synchronization

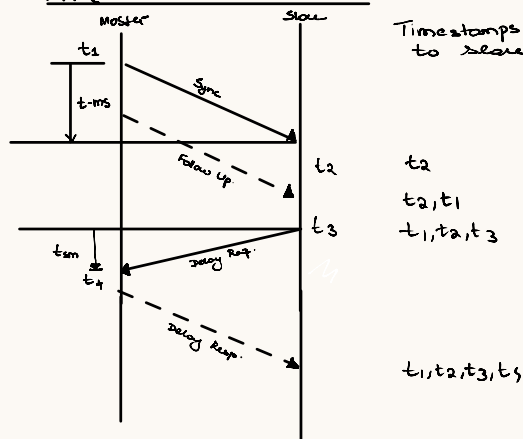
Camera: 30Hz

Robot Controller: 500Hz

Lidar: Continuous

A typical crystal clock drifts about 20 → 100ppm, After 10s, 2 clocks are 1ms apart. Not acceptable for precision sensor fusion & aim control.

PTP (Precision Time Protocol)

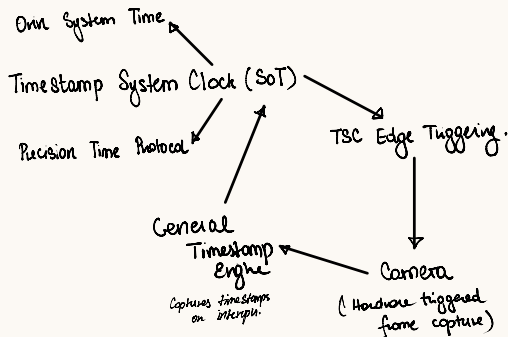


$$\text{Delay} = \frac{(t_4 - t_1) - (t_3 - t_2)}{2}$$

$$\text{Offset} = (t_2 - t_1) - \text{Delay}$$

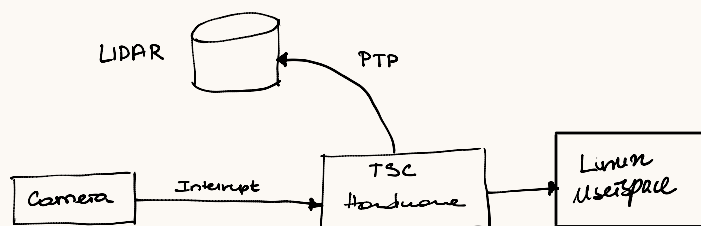
This offset can then be smeared on the clock to bring it in sync

Fun fact: This smearing is also done by Google to handle a leap second.



In essence PTP is a protocol to synchronize multiple device clocks that occur on the same network.

Using ptp4lib we can configure the Jetson to be the grandmaster & synchronize the LIDAR to its own clock & other PTP slaves.



Taking advantage of Jetson Hardware Features, It has a feature to sync the PTP clock with the TSC using PPS from the network card.

This is very useful for applications like this one (sensor fusion & camera frame timestamping)

The Orin also includes the Generic Timestamp Engine which provides hardware timestamping for state changes of specific interrupt lines.

This is used by LibArgus (the Linux API for cameras on Tegra) to obtain SoT / EoT timestamps that are synchronized to TSC.

In this way, all the sensors are synchronized to the same clock domain.

Alternatively, you could trigger camera frames using FSYNC / PPS based on the TSC clock, But you must enable infinite timeouts in the Argus driver & connect to the Jetson's GPIO pins*

I couldn't find good documentation for the e-con Camera so I'm just going over keywords I've found:

The URTE uses its own clock that I can't figure out how to synchronize so what I'd compromise with is taking the timestamp from RTDE & using that to convert time from the robot to PTP time down & vice versa, kind of like this

$t_1 = \text{getTime}$
 $ur = \text{RTDE.getTime}$
 $t_2 = \text{getTime}$

Assume RTDE timestamp taken at $\frac{t_1 + t_2}{2} = e$

$$\text{offset} = \underbrace{(e / 1e9)}_{\text{convert from ns to us}} - ur$$

You can use a 1000 ring buffer to store history of the URTE joint states so you can remember the joint states from when a particular image frame was taken. & since the controller can respond faster than the camera, otherwise if a state is discretely stored, you can interpolate from the 2 nearest states

Occasionally you'll have to re-run this 'manual' synchronization

Synchronization Pseudocode for camera.

```

t1 = SystemTime()
tc = getCameraFrame().ts → Get the next available camera frame.
t2 = SystemTime()

offset = (t1+t2)/2 - tc
    
```

Trigger Generator

```

ts = offset / 1e3; nTs = 1e3
while (True) {
    emitPulse (FSYNC_PIN);
    period += ts * (nTs - 1);
    sleep(period);
}
    
```

Very rough idea to spread out the offset over 1000 frames (a 16s at 30fps), ensuring the camera encounters some clock drift in a > 16s.

5) Resource Monitoring.

Performance Metrics

```

struct SystemHealth {
    // --- Timing & Synchronization ---
    int64_t lidar_ptp_offset_ns; // from ptp4l - target: < 500'000 ns
    int64_t camera_tsc_corr_error_ns; // from correlated_timestamp_driver
    int64_t ur_clock_drift_ppm; // from URClockTracker
    uint64_t last_trigger_jitter_ns; // actual vs intended FSYNC fire time
    uint32_t camera_frames_dropped; // rolling 1s window
    uint32_t lidar_packets_missed; // rolling 1s window

    // --- AI Inference ---
    float inference_latency_ms; // latest TensorRT forward pass
    float inference_latency_p99_ms; // 99th percentile over 10s window
    uint32_t trajectory_buffer_depth; // waypoints ahead of current time
    // < 3 = AI struggling
    uint64_t last_ai_trajectory_ns; // age of most recent trajectory
    bool ai_timeout; // set if > 80ms since last trajectory

    // --- Control Loop ---
    float control_loop_jitter_us; // 500Hz loop period deviation
    uint32_t rtde_write_failures; // failed RTDE packets (rolling 1s)
    bool deceleration_active; // true = AI timeout engaged

    // --- Thermal & Power ---
    float gpu_temp_celsius;
    float cpu_temp_celsius;
    float gpu_power_watts;
    uint32_t gpu_throttle_events; // non-zero = thermal throttle occurred
    // leading indicator before shutdown

    // --- Robot Safety ---
    bool estop_engaged;
    float tcp_velocity_m_s; // must remain < safety_limit
    float joint_torque_max_nm; // highest torque across all joints
    bool protective_stop_active; // UR internal safety violation

    // --- System-Level ---
    float cpu_load_1s[12]; // per-core load
    uint64_t core2_max_latency_ns; // worst RT wakeup latency on isolcpus core
    uint32_t watchdog_misses; // health monitor heartbeat misses

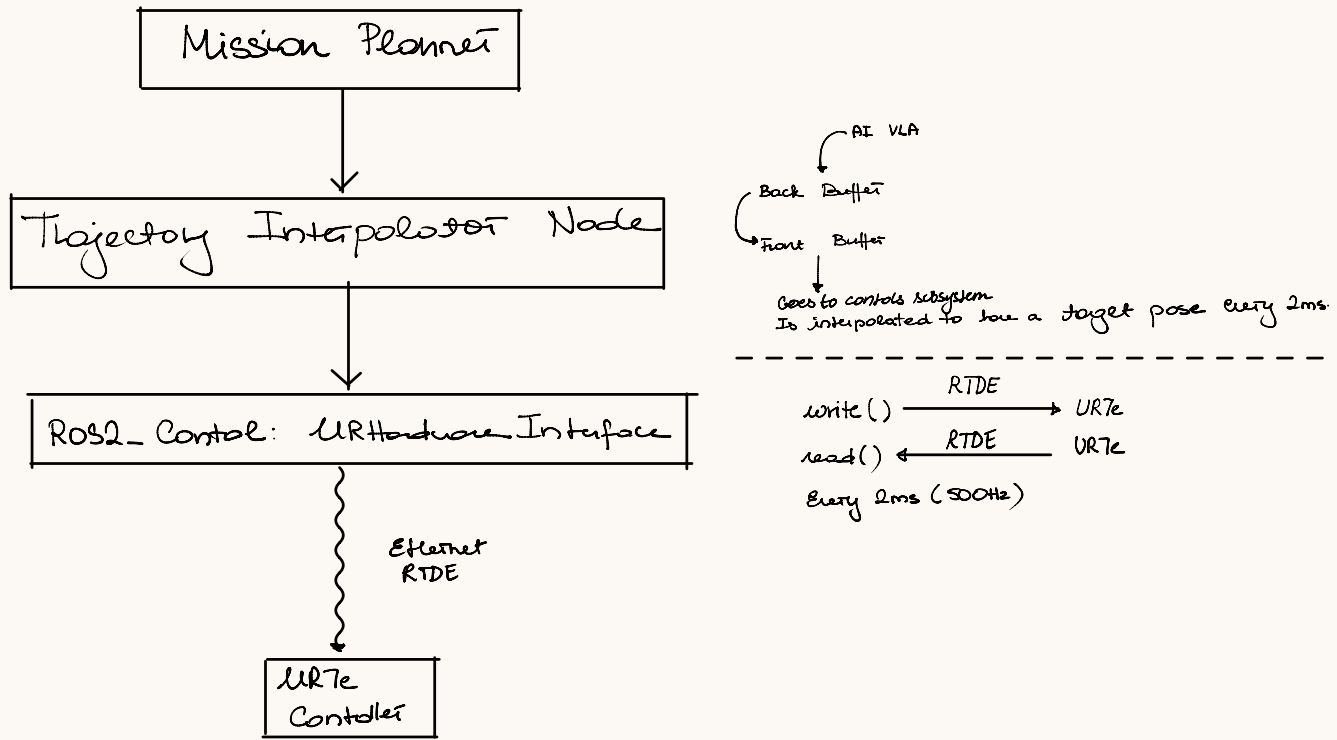
    // --- Derived Status ---
    enum class Status { GREEN, YELLOW, RED, ESTOP } overall_status;
};
    
```

Stoked Knutling)

Action Matrix

Camera Frame Dropped	Warning	Log Event. Tag Current Frame as STALE. Use last valid frame with dead reckoning estimation for 1 cycle. Increment drop counter. If counter > 3, escalate to Error. Pause Inference. Alert mission planner.
Lidar PTP Offset > 5ms	Error	Reject incoming point cloud, Attempt PTP re-sync. Pause trajectory updater. If not resolved in 2s, escalate to CRITICAL & command robot to
Inference Latency > 100ms	Critical	Immediately freeze trajectory buffer at current waypoint. Engage deceleration. Engage trajectory toward safe park pose. Wait for Clear Fault before resuming.
GPU Temp > 85°C	Critical	Reduce GPU clock to min. power state. Drop inference to 10Hz fallback mode (if any). Command robot to request safe park pose. Wait till temp < 75°C for 30s.

4)



Trajectory Interpolator Pseudocode

MAX_WAYPOINTS = 64

front = []

back = []

f_c = 0

b_c = 0

atomic <back> new_trajectory_ready = false;
atomic <write> last_ai_update_ns = 0;

def push(waypts):
 b_c = min(len(waypts), MAX_WPTS)
 back = [... waypts[0:b_c]]
 last_ai_update_ns = now();
 new_trajectory_ready = True;

// Called by controller loop

def maybe_suspend():

 if (new_trajectory_ready == False){
 suspend(front, back);
 f_c = b_c

def interpolate_at(ts):

 for i in range(0, f_c):
 if front[i].ts <= ts && front[i+1].ts >= ts:
 $\alpha = (ts - front[i].ts) / (front[i+1].ts - front[i].ts)$
 return lerp(front[i], front[i+1], α)

 return front[f_c+1]

Core	Task	Priorities	Policy
0	Linux OS ROS2 Middleware Logging & Non RT Tasks	Default	SCHED_OTHER
1	Linux Driver Camera Driver Camera HD Trigger	70 70 80	SCHED_FIFO
2	500Hz RTDE Control Loop [ISOLATED]	99 99	"
3	Health Monitor Watchdog Drift Tracker Trajectory Buffer Manager	50 50 60	"
4-7	Mission Planner API Services	Default	SCHED_OTHER
	Inference	"	"

During startup for each real time thread,
you want to pin the memory pages &
preallocate some stack space

6)



