Atomic Pointer Swap Algorithm

A Triple - Buffer Solution for Real - Time Motor Control

RISE Motion Control Systems

January 2026

Contents

1 Introduction

This document describes the Atomic Pointer Swap Algorithm (APSA), a triple buffer synchronization mechanism designed for the RISE exoskeleton control system. The algorithm enables safe, lock-free communication between a safety-critical EtherCAT control loop running at 1 kHz and a ROS2-based control system, ensuring that motor position data can be exchanged atomically without introducing latency or risking system failure.

2 Problem Statement

The RISE Motion Control system implements a ROS wrapper that functions as a middleware layer between the Control Loop and the EtherCAT Loop. This architecture presents several critical challenges.

2.1 System Architecture

The system consists of three main components:

- **Control Loop**: Subscribes to ROS topics and receives desired motor positions. This component determines the target positions for all motors based on trajectory planning and control algorithms.
- **ROS Wrapper (Rise-Motion)**: Middleware layer that forwards motor positions from ROS to EtherCAT and publishes current motor positions back to the ROS ecosystem for feedback to Control, Trajectory, and other subsystems.
- **EtherCAT Loop**: Executes motor commands at hardware level. This safety-critical loop communicates directly with motor controllers and must maintain precise 1 kHz timing.

2.2 Critical Challenges

The implementation must address three fundamental requirements:

2.2.1 Challenge 1: Guaranteed EtherCAT Loop Stability

The EtherCAT loop operates at 1 kHz (1 ms cycle time) and is safety critical. Any deviation from this frequency, whether running slower than 1 kHz or stopping entirely will cause the exoskeleton to collapse. This creates an absolute requirement:

- The EtherCAT loop must *never* block or wait for data
- Computation within the EtherCAT loop must be minimal (bare necessities only)
- No locks, mutexes, or blocking operations are permitted in the EtherCAT loop

2.2.2 Challenge 2: Frequency Synchronization

The EtherCAT loop's 1 kHz frequency should serve as the master clock for all related processes:

- The Control loop should also run at 1 kHz
- Motor position publishing to ROS should occur at 1 kHz
- All data exchanges must keep pace with the EtherCAT loop without introducing delays

2.2.3 Challenge 3: Atomic Synchronization

Operating at high frequencies (1 kHz = 1 ms cycle time) means that race conditions are highly probable:

- Motor position data is being written by one component while being read by another
- Without proper synchronization, partial writes or reads can occur, leading to corrupted data
- Traditional locking mechanisms (mutexes, semaphores) would violate Challenge 1 by potentially blocking the EtherCAT loop
- The solution must provide atomic data exchange without blocking

3 Solution Overview: Triple - Buffer Architecture

The Atomic Pointer Swap Algorithm addresses all three challenges using a lock-free, triple-buffer architecture. The key insight is that by using three separate memory buffers and atomic pointer swaps, we can allow the Communication thread and the Performance (EtherCAT) thread to work independently without ever blocking each other.

3.1 Core Principle

The algorithm uses three buffers with atomic pointer exchanges:

- **Communication Buffer**: Where new data arrives from ROS or where data is prepared for publication to ROS
- **Atomic Buffer**: An intermediate buffer that serves as a safe handoff point between threads
- **Performance Buffer**: The buffer used by the EtherCAT loop.

By swapping pointers atomically (a single-instruction operation on modern processors), we ensure that:

1. No thread ever sees partially written data
2. The EtherCAT loop never waits or blocks
3. Each thread always has a complete, valid buffer to work with

4 Formal Definition

4.1 State Components

The algorithm defines three states, each containing a pointer to a data buffer:

State	Notation	Description
Communication	c	ROS thread state
Atomic	a	Handoff state
Performance	p	EtherCAT thread state

Each state contains:

- **Pointer**: A pointer variable (c_p, a_p, p_p) that references a memory buffer
- **Memory Buffer**: A memory location (c_{mem}, a_{mem}, p_{mem}) containing motor position data
- **Data Object**: The actual motor position data stored in memory (c_{obj}, a_{obj}, p_{obj})

4.2 Special Performance State Component

The Performance state has an additional component:

- **IO Buffer (p_{IO})**: A dedicated buffer that interfaces directly with the EtherCAT hardware. This buffer represents the actual hardware registers or DMA memory that the EtherCAT master reads from or writes to.

4.3 Null State Semantics

A pointer can be set to `null` to indicate that no data is ready for exchange. This allows threads to check whether new data is available without blocking:

- If $a_p = \text{null}$, there is no new data waiting in the atomic state
- After processing data, threads set their pointer to null to indicate completion

5 Algorithm Description: Send Path

The send path handles forwarding motor positions from the ROS Control system to the EtherCAT hardware. This involves two threads working independently.

5.1 Communication Thread (ROS Subscriber)

5.1.1 Step 1: Receive New Motor Positions

When the ROS topic publishes new desired motor positions:

- Write the new motor positions into c_{mem}
- This write operation is safe because only the communication thread accesses c_{mem} at this point

5.1.2 Step 2: Atomic Swap with Atomic Buffer

Perform an atomic pointer exchange between communication and atomic states:

```
atomic_exchange(c_p, a_p)
```

This single atomic instruction swaps the pointers, making the new data available to the performance thread while giving the communication thread access to the previous atomic buffer (now available for the next write).

5.2 Performance Thread (EtherCAT Loop)

Running at 1 kHz, the performance thread executes every millisecond:

5.2.1 Step 1: Check for New Data

Check if new data is available without blocking:

```
if (a_p != null) { ... }
```

If a_p is null, there is no new data. The performance thread continues with the previous motor positions, maintaining the 1 kHz cycle time without delay.

5.2.2 Step 2: Atomic Swap with Atomic Buffer

If new data is available:

```
atomic_exchange(p_p, a_p)
```

The performance thread now has access to the new motor positions, and the atomic buffer is reset (p_p was null before the swap, so a_p becomes null after the swap).

5.2.3 Step 3: Write to IO Buffer

Copy the motor positions from p_{mem} to p_{IO}:

```
memcpy(p_io, p_mem, size)
```

The EtherCAT master then transmits these positions to the motor controllers. This is the only computationally expensive operation in the performance thread, and it must complete within 1 ms.

5.2.4 Step 4: Reset Pointer

After writing to the IO buffer:

```
p_p = null
```

This signals that the performance thread has completed processing and is ready for the next data exchange.

6 Algorithm Description: Receive Path

The receive path handles reading actual motor positions from the EtherCAT hardware and publishing them to the ROS ecosystem for feedback control. The flow is reversed compared to the send path.

6.1 Performance Thread (EtherCAT Loop)

6.1.1 Step 1: Read from IO Buffer

At each 1 kHz cycle, the EtherCAT master receives actual motor positions from the hardware and writes them to p_{IO}. The performance thread then copies this data:

```
memcpy(p_mem, p_io, size)
```

This ensures that p_{mem} contains the latest motor positions.

6.1.2 Step 2: Atomic Swap with Atomic Buffer

Make the new data available to the communication thread:

```
atomic_exchange(p_p, a_p)
```

6.2 Communication Thread (ROS Publisher)

The communication thread runs independently, checking for new data to publish:

6.2.1 Step 1: Check for New Data

```
if (a_p != null) { ... }
```

If a_p is null, no new data is available. The thread continues without blocking.

6.2.2 Step 2: Atomic Swap with Atomic Buffer

If new data is available:

```
atomic_exchange(c_p, a_p)
```

6.2.3 Step 3: Copy to Local Variable

Copy the motor positions from c_{mem} to a local variable for publishing:

```
local_data = c_mem  // or memcpy if needed
```

6.2.4 Step 4: Reset Pointer and Publish

```
c_p = null
ros_publish(local_data)
```

The motor positions are now published to the ROS topic, making them available to the Control, Trajectory, and other subsystems.

7 Pseudocode Implementation

7.1 Initialization

```
// Allocate three data buffers
c_mem = allocate_buffer()
a_mem = allocate_buffer()
p_mem = allocate_buffer()
p_io = get_ethercat_io_buffer()

// Initialize pointers
c_p = &c_mem
a_p = null       // Atomic starts empty
p_p = &p_mem
```

7.2 Send Path Implementation

Communication Thread (ROS Subscriber Callback):

```
function on_motor_command_received(new_positions):
    // Write new positions to communication buffer
    write_to_buffer(c_p, new_positions)

    // Atomically swap communication and atomic pointers
    atomic_swap(&c_p, &a_p)
```

Performance Thread (1 kHz EtherCAT Loop):

```
function ethercat_cycle_send():
    // Check if new data is available
    if a_p != null:
        // Atomically swap performance and atomic pointers
        atomic_swap(&p_p, &a_p)

        // Copy new positions to IO buffer
        memcpy(p_io, p_p, data_size)

        // Signal completion
        p_p = null

    // EtherCAT transmission happens automatically
    ethercat_send_frame()
```

7.3 Receive Path Implementation

Performance Thread (1 kHz EtherCAT Loop):

```
function ethercat_cycle_receive():
    // EtherCAT reception happens automatically
    ethercat_receive_frame()

    // Copy actual motor positions from IO buffer
    memcpy(p_p, p_io, data_size)
```

```
    // Make data available to communication thread
    atomic_swap(&p_p, &a_p)
```

Communication Thread (ROS Publisher):

```
function publish_motor_feedback():
    // Check if new data is available
    if a_p != null:
        // Atomically swap communication and atomic pointers
        atomic_swap(&c_p, &a_p)

        // Copy to local variable for publishing
        local_positions = read_from_buffer(c_p)

        // Signal completion
        c_p = null

        // Publish to ROS
        ros_topic.publish(local_positions)
```