Scalable Decentralized Asynchronous System Design: Camera–Monitor Architecture

Table Of Contents

Introduction

Project Description

This project presents the design of a scalable decentralized and asynchronous Camera–Monitor system that operates using shared memory. The system consists of two independent agents, a camera and a monitor, that coordinate only through shared-memory flags and rules. Each agent runs at its own clock frequency and continues functioning even if the other is removed. With only minor modifications, the same principles can be extended to many more agents, making the design naturally scalable. The first part of this project focuses on the baseline case of one camera and one monitor, while the later sections extend the same principles to multi-agent scaling.

Motivation

This project was created to demonstrate how simple decentralized rules, when combined with a shared resource, can produce robust and scalable system behavior. Instead of relying on a central controller, each agent (camera and monitor) operates independently, coordinating only through shared-memory flags. This approach highlights key principles of system design: simplicity, fault tolerance, and scalability.

Concepts Demonstrated

Main Concepts in the Design

- **Decentralized Asynchrony:** agents operate independently at different clock rates, without a master controller or global timing.
- **Shared-Memory Communication:** coordination occurs only through flags, pointers and metadata, avoiding direct messaging.
- **Scalability**: the same mechanism extends naturally to multiple agents with minimal design changes.

Additional Concepts

- **Fault Tolerance and Low Coupling:** the system keeps functioning even if one agent fails, and each agent can be upgraded or replaced independently.
- **Overlap Optimization:** in certain configurations, reads and writes can proceed in parallel, reducing latency and improving efficiency.
- **Future Extensions:** this design offers only a glimpse of the many possible improvements and advanced variations, with a taste of them presented later in the document.

Together, these concepts form the foundation of the system. The next sections describe how they are implemented in the Camera–Monitor architecture.

Theoretical Perspective: Viewing Time as Discrete

Although time in the real world is continuous, for the purpose of analyzing digital systems it is often more practical to view time as a sequence of discrete moments. By dividing time into sufficiently small intervals, we can assume that the activity within each interval is negligible and instead describe the system as a timeline of events: “what happened at each moment.”

This perspective is especially useful for decentralized asynchronous systems like the one presented here. Since there is no central controller, the interaction between agents (camera, monitor, and shared memory) must be validated step by step. By modeling the system in discrete time, we can clearly track the state of each agent at each moment, verify how they interact, and confirm that the system functions correctly without any master controller.

For example, the table below illustrates a simplified timeline:

Time	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**…**
Camera		on	write		write		write			off	
Monitor						on	read		read		
Shared Memory			Input		input		input output		output		

It is important to emphasize that this discrete timeline exists only for observation and analysis. The subsystems themselves are unaware of time or of each other’s activity – they simply follow shared-memory flags and local rules. Each can run at its own frequency, independent of the others, and can even join or leave the system at any moment without disrupting overall operation.

This discrete perspective makes it possible to analyze and verify decentralized asynchronous behavior without requiring a central authority.

System Technical Description

Note 1: The specific hardware settings and parameters may vary between systems, and even different agents within the same system may operate under different configurations. These variations do not affect the core principles of the design. Throughout this document we use a camera and a monitor as concrete examples, but they could just as well be replaced or combined with other types of agents that follow the same rules. Different configurations can be combined, and the system will still function correctly because the rules of communication and data transfer remain the same. Throughout this section we present, in blue text, concrete examples of our system's parameters (such as resolution, sensor type, or display size) to provide a tangible reference, but these are illustrative only and not essential to the concept. Transfer times are also only examples — in a decentralized and asynchronous design the system works regardless of timing, which mainly influences buffer size and communication limits (see "Further Topics To Explore"). In practice, additional optimizations and improvements can also be introduced, some of which are briefly illustrated in the appendices.

Note 2: The overall structure of our system is shown in Figure 1 – Systems Diagram.

Shared Memory

General

The shared memory serves as the central storage of the system, accessible to both the camera and the monitor. Each agent interacts with it under different permissions (see "Table 1: Memory Map").

Hardware

Main Memory

The main buffer is a continuous memory block of fixed size, used as the central storage area of the system.

- The **camera** writes photo blocks into it (the block size varies).
- The **monitor** reads photo blocks from it.
- Reads and writes are aligned to the system's word size (4 bytes in our design).
- The buffer size should be chosen according to the system's design considerations (see "Further Topics for Exploration").
  In our system, the buffer size is set to 10 times the maximum image block size (see "Image Block").

The main buffer operates linearly with start and end pointers, so no random access is required. However, when new writes cut into existing data (overrides), the leftover parts are marked as "junk blocks". This creates fragmentation inside the buffer, but since all reads are sequential, the monitor simply encounters and skips junk in order.

Pointers

Two pointers control access to the buffer. Each is n bytes wide (depending on the design) and stores an address.

- **Start pointer:** marks the last byte of the most recently read block.
- **End pointer:** marks the last byte of the most recently written block.

These pointers are updated by the agents and form a key element of the communication mechanism.

Shared Flags

These flags are updated by the agents and form a key element of the communication mechanism as well.

- **WRITE:** set by the camera while writing to the buffer, cleared when writing is complete.
- **OVERRIDE:** set by the camera if the new write CAM_OPERATION overwrites still-valid data, cleared when writing is complete.
- **READ:** set by the monitor while reading from the buffer, cleared when reading is complete.

Concurrent Access Guard Mechanism

Whether pointers are advanced before or after the operation, the same risk exists: multiple agents may try to update the pointers in the same cycle, or one may arrive while another has not yet finished writing its metadata. To prevent this, the memory system applies a tiny hardware guard: if two requests collide, one is granted, and the others are denied (to retry later). This ensures only one agent can claim a block at a time, avoiding half-updated states. The arbitration happens only at the memory access level, so the system remains decentralized and asynchronous.

Camera

General

The camera is an independent agent responsible for capturing photos, attaching metadata, (and in our system EDC), and transferring complete image blocks into the shared memory buffer. It operates through the shared pointers (Start/End) and flags (READ / WRITE / OVERRIDE) in a fully asynchronous manner, without direct coordination with the monitor.

Hardware:

Setting

The setting defines the physical environment and positioning in which the camera operates, including its distance from the scene and its mechanical mounting.

- Mounted 1 m from a 10 m × 10 m scene board.
- Positioned on a two-dimensional rail system that enables horizontal and vertical movement at a constant speed of 10 cm/s.

Lens

The lens determines the camera's optical characteristics, including magnification and field of view.

- Equipped with a fixed-zoom lens that provides a field of view (FOV) of 48 cm × 64 cm at a distance of 1 m.

Sensor

The sensor is the core element that captures raw image data, defined by its pixel grid, resolution modes, and color formats.

- **Fixed grid:** 192 × 256 px (49,152 samples).
- **Supports 3 binning modes:** (Appendix 1) combining neighboring pixels, reducing resolution and image size:
    1. **1 × 1 (no binning):** 192 × 256 = 49,152 samples.
    2. **2 × 2 binning:** 96 × 128 = 12,288 samples.
    3. **4 × 4 binning**: 48 × 64 = 3,072 samples.
- **Supports 2 color modes:**
    1. **RGBA8888:** 4 bytes per pixel (full color).
    2. **GRAY8:** 1 byte per pixel (grayscale intensity).

Note: within the full system setting, a maximum-resolution photo taken by the camera corresponds to 0.25 × 0.25 cm of the real world per pixel.

Rail Engine

The Rail Engine moves the camera according to the controller's instruction.

- **Fixed speed:** 10 cm/s, corresponding to 0.1 cm per cycle at a 100 Hz clock.

BIN_AND_COL flag (1 byte)
Specifies resolution and color mode for the photo request.

- Checked only if CAM_OPERATION = Photo Request.
- Value is determined by user switches (resolution + color/grey).

Value Table:

Color/Resolution	192 × 256 px	96 × 128 px	48 × 64 px
Color (RGBA8888)	00000101	00000011	00000001
Grey (GRAY8)	00000100	00000010	00000000

CAM_OPERATION flag (1 byte)
Determines whether the user has requested movement, a photo or no action.

- Checked by the controller at the start of every cycle.
- When the user presses a button, a circuit updates this flag. Once written, the value remains unchanged until the Controller resets it to 0.

Value Table:

CAM_OPERATION	None	Up	Down	Left	Right	Photo Request
Value (binary)	00000000	00000001	00000010	00000011	00000100	00000101

Internal Buffer
The camera maintains a single internal frame buffer in the size of the maximum Image Block (in our system ~192 KiB).

- **Capture:** Used by the Image Processing Unit (IPU) to assemble a complete image block. The IPU streams metadata, raw pixel data, and CRC-32 into this buffer until the block is fully written and verified.
- **Transfer:** Once the Capture Buffer contains a complete valid block, the Camera Control Unit transfers the block into the Shared Memory, updates the End pointer, and clears the buffer for the next capture.

This mechanism reduces the risk of partially written or corrupted frames in shared memory, since only complete and validated photo blocks are normally transferred. However, if an error occurs during transfer from the internal buffer to shared memory, the block may still be written incorrectly and the Camera Controller will handle it according to the system's protocol.

Image Processing Unit (IPU):
The Image Processing Unit is the dedicated module responsible for converting raw sensor data into a complete image block ready for transfer.

- Activated by the Control Unit when CAM_OPERATION = Photo Request and the request is validated.
- Connected read-only to the sensor and write-only to the camera's internal buffer.

During activation, the IPU performs the following sequence:

1. Receives metadata from the Control Unit and writes it into the internal buffer.
2. Reads raw pixel data from the sensor grid (192 × 256) and applies binning and color formatting according to the selected mode.
3. Streams the processed data into the internal buffer while simultaneously updating a CRC-32 checksum.
4. Appends the 4-byte CRC-32 at the end of the buffer once the transfer is complete.
5. Returns control to the Control Unit, reporting whether the operation succeeded.

This separation ensures that while the IPU is processing and filling the internal buffer, the Camera Control Unit remains free to manage system logic and shared-memory coordination.

Control Unit & Clock
The camera is managed by a main controller operating at a fixed internal clock of 100 Hz.

- At the start of each cycle, the controller checks the internal flags and executes CAM_OPERATIONs accordingly.
- At the end of each CAM_OPERATION, the controller resets the CAM_OPERATION flag to zero.

Control Unit actions in relation to CAM_OPERATION flag

- **CAM_OPERATION = None:** no action is performed.
- **CAM_OPERATION = Up/Down/Left/Right:** the controller activates the Rail Engine to move the camera within the same cycle, then resets the CAM_OPERATION flag to zero.
- **CAM_OPERATION = Photo Request:** see "Read and Write requests".

Note: While a photo is being captured and transferred into shared memory, the CAM_OPERATION flag remains locked. During this period, the camera does not accept new requests until the process is complete.

Photo Request Timing

- **IPU I/O overhead:** 10 cycles.
- **IPU transfer rate:** 500 bytes (≈ 0.5 KB) per cycle, equivalent to 50 KB/s.
- **Data transfer rate:** 500 bytes (≈ 0.5 KB) per cycle, equivalent to 50 KB/s.

Display Sequence and Request Timing:

Write process
When a photo request is issued, the IPU assembles the image in the internal buffer and, once complete, transfers it into shared memory.

Timing

- **IPU I/O overhead:** 10 cycles.
- **IPU transfer rate:** 500 bytes (≈ 0.5 KB) per cycle, equivalent to 50 KB/s (Memory read is assumed 10 × faster than camera write throughput).

- **Data transfer rate to shared memory:** 1000 bytes (≈ 1 KB) per cycle, equivalent to 100 KB/s

Monitor

General

The monitor is an independent subsystem whose role is to read blocks from the shared memory and display them. It operates with the same pointers (Start/End) and flags (READ / WRITE / OVERRIDE) in a fully asynchronous way, without direct coordination with other agents.

Note: The last valid image remains displayed until it is replaced by a new one.

Hardware

Display
Fixed-size display of 192 × 256 px, operating at 92 PPI (standard HD).

- **Physical Dimensions:**
  1. **Width**: $\frac{256}{92}$ inches ≈ 2.78 inches
  2. **Height**: $\frac{192}{92}$ inches ≈ 2.09 inches

Note: in our project, at maximum resolution each pixel of the camera corresponds to 0.25 × 0.25 cm of the real world. Because the monitor resolution (192 × 256 px) matches the camera resolution, the image is displayed at the same scale. Thus, one monitor pixel also represents 0.25 × 0.25 cm of the original scene.

MON_OPERATION flag (1 byte)
This flag indicates whether the user has requested a new photo to be read from shared memory and displayed.

- Checked by the controller at the start of every cycle.
- Determines whether the user has requested next photo to monitor.
- When the user presses a button, a circuit updates this flag. Once written, the value remains unchanged until the Controller resets it to 0.

Value Table:

MON_OPERATION	None	Read
Value (binary)	00000000	00000001

Internal Double Buffer
This buffer system ensures that one frame can be displayed while the next is being prepared, preventing the user from ever seeing a half-updated image.
The monitor maintains two internal frame buffers, each in the size of the maximum Image Block (in our system ~192 KiB).

- **Current Buffer:** actively used by the Display Controller to refresh the screen.
- **Next Buffer:** When the next buffer is fully updated and verified (metadata + data + CRC-32), the Control Unit swaps buffers: the next becomes current, and the old one is cleared for reuse. This ensures the user never sees a half-updated frame.

Display Controller
This controller drives the monitor panel by reading pixel data from the buffer and sending it row by row to the display.

- Activated by the Monitor's Control Unit when MON_OPERATION = Read and the request is validated.
  Connected read-only to the Internal Double Buffer.
- Reads pixel data sequentially from memory and refreshes the panel.
- Works independently of the Monitor Control Unit – its only input is the "current buffer" pointer.

This separation ensures that while the Control Unit is preparing the next frame, the Display Controller keeps the visible image stable.

Monitor Control Unit & Clock
This unit manages the monitor's operation, running on a 60 Hz internal clock.

- At the start of each cycle, the Control Unit checks the shared memory flags and the Start/End pointers.
- If MON_OPERATION = Read and a new photo block is available in the buffer (Start ≠ End), it initiates the reading procedure.
- At the end of each operation, the Control Unit resets the MON_OPERATION flag to zero.

Control Unit actions in relation to MON_OPERATION flag:

- **MON_OPERATION = None:** no action is performed.
- **MON_OPERATION = Data:** see "Read and Write requests".

Note: While reading photo to internal buffer, the MON_OPERATION flag remains locked for the duration of the process. During this time, the Monitor does not accept additional MON_OPERATIONs until the transfer is complete.

Display Sequence and Request Timing:

Read process
The Control Unit first transfers data from shared memory into the internal buffer, and once this is completed, the Display Controller reads from the buffer and updates the screen.

Timing

- **I/O overhead (time for the Monitor Control Unit to access shared memory):** 10 cycles.
- **Monitor Control Unit data transfer rate:** 5,000 bytes (≈ 5 KB) per cycle, equivalent to 0.5 MB/s (Memory read is assumed 10 × faster than camera write throughput).
- **Display Controller read rate:** 2,500 bytes (≈ 2.5 KB) per cycle.

System Diagram and Memory Map

Systems Diagram

The behavior of agents (camera and monitor) depends on many configuration choices.
To keep the description concrete, this diagram illustrates one specific setup:

- 1 Camera and 1 Monitor.
- Overwrite enabled.
- No overlap (read-while-write / write-while-read disabled).
- Method 2 (commit-on-success, pointer updated last).
- Single attempt per read: if data is invalid, no retry is performed.

This configuration serves as the baseline and provides a simple reference point for understanding the core logic. Later appendices extend this baseline to more advanced cases.
Access permissions corresponding to this setup are summarized in Table 1 (Memory Map).

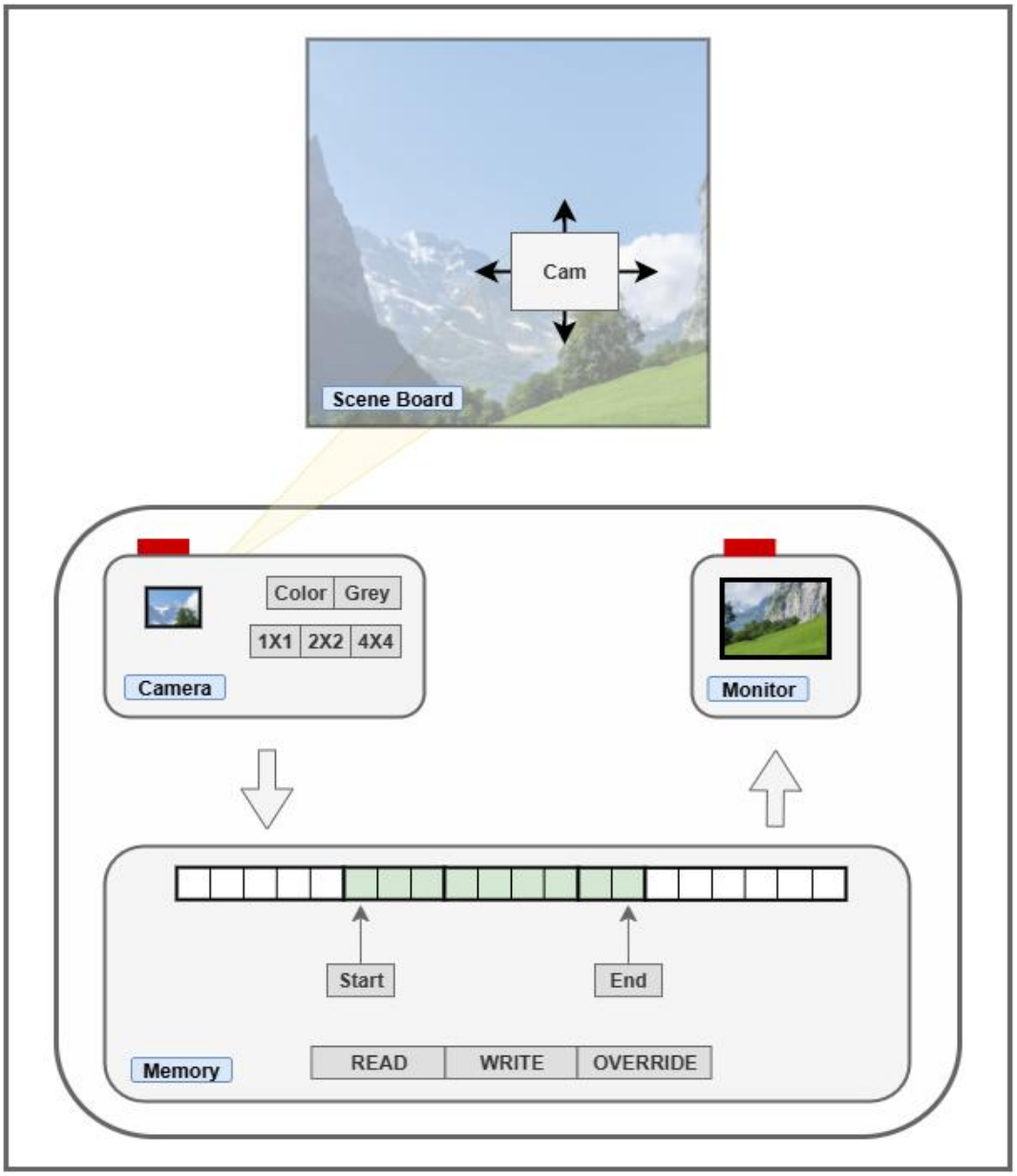


Figure 1: System Diagram

Memory Map

In addition to the system diagram, it is important to define the access permissions of each agent to the shared resources. Table 1 summarizes these access rights.

Memory/Agent	Camera Controller	Monitor controller
Shared Buffer	Write-only	Read-only
Start pointer	Read + Write	Read + Write
End pointer	Read + Write	-
READ flag	Read-only	Write-only
WRITE flag	Write-only	Read-only
OVERRIDE flag	Write-only	Read-only
CAM_OPERATION	Read + Write	-
MON_OPERATION	-	Read + Write
BIN_AND_COL	Read-only	-

Table 1: Memory Map

Image Block

Structure

An image block is the fundamental data unit exchanged between the camera and the monitor. In our system, each block is composed of three sub-blocks: metadata, image data, and CRC-32.

metadata		image data	CRC-32
Type	Junk		
1 byte	3 bytes	0 – 192 KiB	4 bytes

metadata

The metadata is stored in the first four bytes of each image block:

- **First byte:** represents the Type of Image (by BIN_AND_COL).
- **Next three bytes:** represent the size of junk data (0 if none).

Color/Resolution	192 × 256 px	96 × 128 px	48 × 64 px
Color (RGBA8888)	00000101	00000011	00000001
Grey (GRAY8)	00000100	00000010	00000000

Note: if read-while-write or write-while-read is enabled, additional status bits are appended to the metadata (see “Overlap”).

Image data

The image data is created by the IPU and its size is determined from the BIN_AND_COL value:

Color/Binning	192 × 256 px	96 × 128 px	48 × 64 px
Color (RGBA8888)	196,608 bytes 192 KiB	49,152 bytes 48 KiB	12,288 bytes 12 KiB
Grey (GRAY8)	49,152 bytes 48 KiB	12,288 bytes 12 KiB	3,072 bytes 3 KiB

CRC-32

The final 4 bytes contain a CRC-32 checksum, used as an error-detecting code to verify data integrity during transmission and storage. CRC-32 reliably detects common errors such as single-bit flips, burst errors, or random corruption. It does not correct errors (as ECC codes do), it only signals that corruption has occurred (see “Further Topics To Explore”).

Pointer Handling, Override Check and Read Failures

Cyclic Buffer Logic

The shared buffer is implemented as a circular memory structure (ring buffer), where all addresses are computed modulo the total buffer size.

- **Write (Camera):** Moves End pointer to: $(\text{Current End} + \text{Image Block Size}) \bmod (\text{Buffer Size})$.
- **Read (Monitor):** Moves Start pointer to: $(\text{Current Start} + \text{Image Block Size}) \bmod (\text{Buffer Size})$.

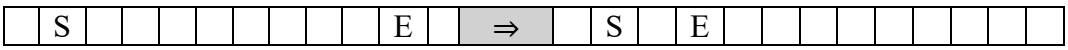
If an addition crosses the buffer boundary, the new address wraps around to the lower addresses.

Override

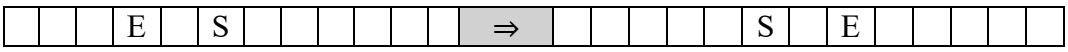
Override Detection

An override occurs when a new write advances the End pointer beyond the Start pointer, indicating that unread data in the buffer will be lost. This situation can arise in three distinct cases:

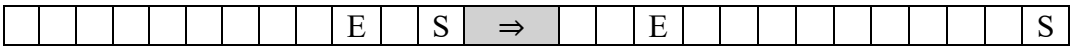
- **Start < End, End wraps around:** If the new End address after wraparound is greater than Start.



- **Start > End (End already wrapped):** If the new End becomes greater than Start.



- **Start > End (Another wrap):** If the new End is lower than the current End (another wrap).



Override Process:

When an override is detected, the Camera Controller resolves it as follows:

1. It sums the sizes of the blocks to be overwritten until this sum exceeds the size of the new block.
2. The new image block is written into the buffer. Fully covered blocks are directly overwritten.
3. If the new block ends in the middle of an existing block (the common case), that block is replaced by a junk block. The junk size equals the total size of the overwritten blocks minus the size of the new block, and the junk block is written immediately after the new block.
4. The End pointer is updated to the new position.
5. The Start pointer is advanced to the end of the junk block.

This mechanism ensures that the monitor can detect and skip corrupted blocks by identifying them as junk.

Example:

Assume the Camera Controller receives a Photo Request of size 5 blocks.
In this case, the camera overwrites three full blocks and partially overwrites a fourth block. In this case, the Start pointer must be advanced to the position immediately following the junk block created by the partial overwrite:

1. $\text{Sum} = (1 + 2 + 1 + 2 = 6)$
2. Write the new block of size 5.
3. Write a junk block of size $(\text{Sum} - \text{New Block Size}) = (6 - 5) = 1$
4. Move both End and Start to their updated positions.

Read Failure

If a read request occurs when Start = End, the buffer is empty and no valid data is available.

In the cyclic layout, any other relation between Start and End (Start > End or End > Start) still represents valid data – only equality indicates that the buffer is empty.

Note: In our design, read and write operations are not symmetric. Overrides are allowed during writing, but the monitor is never permitted to read junk data. This restriction follows from the assumption that every block begins with valid metadata, and encountering junk would violate that assumption. Moreover, reading junk data is generally unhelpful, since it cannot produce a meaningful image.

Read and Write Request

The handling of read and write requests follows the rules described in the section Reading & Writing – Two Methods. In our baseline example, this section illustrates how these rules are applied in practice our configuration: one camera, one monitor, overwrite enabled, no overlap (read-while-write / write-while-read disabled), Method 2 (commit-on-success), Deny-and-continue, and no retries.

Note 1: the specific handling sequence depends on the chosen configuration, but this section emphasizes the main idea.

Note 2: For readability, the rest of this section is presented in standard text rather than highlighted, even though it describes our specific baseline configuration.

Write Requests (CAM_OPERATION = Photo Request)

When a photo request is issued, the camera evaluates the BIN_AND_COL flag to determine the image size and checks whether an override will occur.

The resulting behavior is summarized in the decision table below:

OVERRIDE / READ	Read = Off	Read = On
No	Write (Regular)	Write (Regular)
Yes	Write (Override)	Deny (active read in progress)

The corresponding actions are:

- **No override expected (regular write):**
  1. Set WRITE = 1.
  2. Trigger the IPU to capture the photo. Once complete, write the image block into shared memory after the current End.
  3. Advance End to the end of the new block.
  4. Set WRITE = 0 and reset CAM_OPERATION.
- **Override expected and READ = ON:**
  1. Deny the photo request (cannot override during an active read).
  2. Reset CAM_OPERATION.
- **Override expected and READ = Off:**
  1. Set WRITE = 1 and OVERRIDE = 1.
  2. Trigger the IPU to capture the photo. Once complete, write the image block into shared memory after the current End.
  3. Advance End to the end of the new block.
  4. If the new write cuts through existing blocks, append a junk block.
  5. Advance Start to the end of the junk block.
  6. Set WRITE = 0, clear OVERRIDE, and reset CAM_OPERATION.

Read Requests (MON_OPERATION = Read Request)

When the monitor initiates a read, it first checks if the buffer is empty. If End = Start (regardless of WRITE), no valid data is available. In this case:

1. Deny the read request.
2. Reset MON_OPERATION.

Otherwise, the monitor evaluates the metadata (including BIN_AND_COL and the junk field) to determine whether the block is valid. If the block is invalid, it is skipped. If it is valid, the monitor derives the image size and then checks the OVERRIDE flag to determine whether an override is in progress.

the resulting behavior is summarized in the decision table below:

OVERRIDE / WRITE	Off	On
No	Read (Regular)	Read (Regular)
Yes	Can’t happen	Deny (override in progress)

The corresponding actions are:

- **No override active:**
  1. Set WRITE = 1.
  2. Transfer the block to the monitor’s internal buffer, verify the CRC, and notify the controller that valid data is available.
  3. Advance Start to the next block.
  4. Set READ = 0, Reset MON_OPERATION.
- **Override active:**
  1. Deny the read request (unsafe during override).
  2. Reset MON_OPERATION.

Together, these procedures ensure that read and write operations remain consistent, preventing partial transfers or data being written while it is still being read, while maintaining the simplicity of the baseline configuration.

Per-Request Handling Policy (wait vs. deny)

Overall

When a request is premature or unsafe – for example:

- A write is requested while the next block is still being read (overlap disabled).
- A write is requested while the next block contains valid data but override is disallowed.
- A read is requested while the next block is still being written.
- A read is requested when the buffer is empty.
- A write is requested while camera busy or a read is requested while monitor busy.

Two local policies are available.

Wait-until-valid (blocking)

The agent holds its operation (no pointer movement, internal OPERATOIN flag stays on), and periodically re-checks the condition (an optional timeout may abort the wait).

- **Read examples:** if BEING_WRITTEN = 1 or an override is in progress, keep READ = 0, MON_OPERATION = 1, leave Start unchanged, and re-check until the block becomes valid.
- **Write examples:** if BEING_READ = 1 and override is disallowed, keep WRITE = 0, leave End unchanged, and re-check until the region is free.

Deny-and-continue (non-blocking):

The agent immediately denies the request and clears its operation flag, pointers remain unchanged and the request may be issued again later.

- **Read examples:** empty buffer (End = Start) or BEING_WRITTEN = 1, deny read, reset MON_OPERATION.
- **Write examples:** BEING_READ = 1 with no override, deny write, reset CAM_OPERATION.

Note: In our configuration, we use deny-and-continue (for simplicity). Systems may enable wait-until-valid (with or without a timeout) as an implementation option, this changes timing only and does not affect correctness of the rules.

Reading & Writing – Two Methods

Overview

Two alternative methods exist for updating the shared buffer pointers. Each has distinct advantages and disadvantages, which are explained below.

Note: With the added metadata described below, both Method 1 and Method 2 support multi-agent scaling (see "Scaling"). The essential trade-off is that Method 1 (pointer-first) allows overlap – a block can be read while still being written, or written while still being read ("see Overlap Techniques"), while Method 2 (commit-on-success) instead provides safer retries by waiting until the block is complete before advancing pointers (see "Pre-Request Handling Policy"). Both approaches scale naturally to multiple readers and writers, but Method 1 emphasizes maximum concurrency, while Method 2 emphasizes reliability.

Method 1 – pointer-first (advance, then operate):

In this method the Start/End pointer is advanced first, and only then the agent begins its Read/Write operation. The block is considered "owned" immediately after the pointer moves.

Metadata

Because the pointer alone does not guarantee that the block is ready or valid (for example, the camera may try to write onto a block that is still being read, while the Start pointer already shows we moved to the next block, or the monitor may try to read a block that is still being written, while the End pointer already advanced), additional status bits must be stored in each block's metadata.

- **Bit 0:** Color/Grey.
- **Bit [1:2]:** Binning.
- **Bit 3:** Valid (1 = Valid, 0 = Junk).
- **Bit 4:** BEING_READ (1 = currently being read).
- **Bit 5:** BEING_WRITTEN (1 = currently being written).

7	6	5	4	3	[0:3]
0 (const)	**0 (const)**	**BEING_WRITTEN**	**BEING_READ**	**Valid**	**BIN_AND_COL**

Each agent updates this metadata immediately after advancing its pointer:

- The camera writes the full 4-byte metadata header and sets BEING_WRITTEN = 1. This guarantees the monitor can see the full block structure.
- The monitor reads the metadata header first, and if BEING_WRITTEN = 1, it knows the camera is still in progress.

Both clear their respective flags (BEING_READ / BEING_WRITTEN) once finished, and the camera sets Valid = 1 when the block is fully written.

Global READ/WRITE flags

In the single-camera/single-monitor design, global READ and WRITE flags (combined with Start and End) were used to signal when a block was busy. In the pointer-first method, these global flags are no longer required: the pointers move immediately, and each block's metadata indicates whether it is currently being read or written.

- The camera just checks the BEING_READ flag before writing.
- The monitor checks the BEING_WRITTEN flag before reading.

Logic

- **Monitor (Read):** On a valid request, the monitor advances Start to the next block and then begins reading.
- **Camera (Write):** On a valid request, the camera advances End (and Start if overriding and creating a junk block), then begins writing.

Pros

- Immediate visibility: other agents instantly know which regions are busy
- Simple concurrency, especially when scaling to multiple cameras and monitors.

Cons

- If an operation fails, the pointer may already point to invalid data. Rollback is complicated and may require junk blocks.
- Requires metadata flags in every block to avoid conflicts between reading and writing.

Method 2 – Commit-on-success (operate, then advance)

In this method, the read or write operation is completed first, and only then is the Start or End pointer advanced.

Logic

- **Monitor (Read):** Completes the read fully, verifies CRC, and only then advances Start. If the read fails and a Retry Policy is enabled, Start remains unchanged and the monitor can attempt the read again.
- **Camera (Write):** Completes the write fully, verifies CRC, and only then advances End (and Start if an override occurs). If the write fails and a Retry Policy is enabled, both pointers remain unchanged and the camera can retry.

Retry Policy

On failure, the Start and End pointers remain unchanged, and the operation can be retried on the same block.

- If the monitor fails to read, Start does not advance, so the same block can be attempted again.
- If the camera fails to write, End does not advance (and Start does not move in the case of override), so the same write can be attempted again.

Pointers are only advanced after a successful operation. If the policy aborts after N attempts, the block is marked junk and skipped on the next access, thereby preventing infinite retry loops.

Pros

- Reduced fragmentation: since pointers advance only on success, most failures do not create junk blocks. Junk is introduced only if the retry policy aborts after N attempts.
- Higher read reliability: if a read fails, Start remains unchanged, allowing retries on the same block and increasing the overall success rate of reads.
- Simplifies error handling, since operations are either fully committed or ignored.

Cons

- Other agents can't see which blocks are "in flight" unless a reservation/busy mechanism is added.
- Lower throughput, since space is only released after full success.

Overlap Techniques

Overview

This technique permits the camera to begin overriding a block even while it is still being read by the monitor, and likewise allows the monitor to begin reading a block while it is still being written by the camera.

Note: Overlap is only possible with Method 1 (pointer-first). In Method 2 (commit-on-success), pointers move only after the operation succeeds, so partial blocks are never exposed and overlap cannot occur.

Claim & Flags

Immediately after advancing its pointer, the agent updates the block header metadata to claim the block.

- **Camera**: writes the full 4-byte metadata header and sets BEING_WRITTEN = 1. Once the payload and CRC are fully written, it clears BEING_WRITTEN and sets Valid = 1.
- **Monitor**: reads the metadata header and sets BEING_READ = 1 when it begins consuming the block. Once the read and verification are complete, it clears BEING_READ.

Thanks to the Concurrent Access Guard, there is no risk that another agent accesses the same block while the metadata update is still in progress.

Read-While-Write

If the monitor encounters a block where BEING_WRITTEN = 1, it may still begin reading. However, it must not advance faster than the camera's write pace. This enables the monitor to access data as soon as it streams in, thereby reducing latency.

Con: if the camera fails mid-write, the monitor may have consumed a partial or invalid block, which must be discarded.

Write-While-Read

If the camera encounters a block where BEING_READ = 1, it may still begin writing. But it must not advance faster than the monitor's read pace. This allows the camera to reuse blocks earlier, also reducing latency.

Con: if the monitor fails mid-read, the overwritten block is lost and retry is impossible.

Pacing Logic

In both cases, the active agent must throttle itself to the slower peer. If it detects the other side is still busy, it assumes the worst case (that the peer just started) and limits its pace accordingly.

- If the peer transfers data at a slower pace, the active side must throttle itself down to match, to avoid overruns.
- If the peer transfers data at a faster pace, the active side can continue at its normal rate, since the peer will be able to keep up.

Scaling

Basic Scaling – multiple agents, one of a kind active at a time

Assume we want to scale the system to **n** cameras and **m** monitors.
If we allow only one camera and one monitor to operate at a time, then no structural changes are needed: all cameras and monitors receive the same access rights as in the single-camera/single-monitor system, and they synchronize via the shared pointers, flags and metadata. The benefit is that multiple components can connect to the buffer, although only one pair is active at any given time.

Advanced Scaling - multiple agents, multiple active at a time

The real power of this design emerges when we allow multiple reads and writes to take place at the same time. Thanks to decentralized coordination through shared pointers, flags, and metadata, the system can safely let many agents use the same buffer concurrently. The rules apply uniformly, independent of agent type, configuration, clock rate, or the moment they join or leave, as long as they respect the shared protocol. This transforms the design from a simple pairwise connection into a scalable multi-agent architecture, where strong behavior is achieved with minimal logic.
Among the possible solutions, we focus here on the pointer-first approach. Rather than describing the full protocol, we outline the core principle and show why it is effective with very simple logic.

Basic Multiple Reads and Writes (With Override)

With the pointer-first approach, each agent immediately advances its pointer when it starts operating:

- **Writes:** as soon as a camera begins writing, the End pointer advances to reserve space. Other cameras see that region as occupied and continue writing after it. Because the pointer moves before the payload is written, no two writers can collide on the same region.
- **Reads:** as soon as a monitor begins reading, the Start pointer advances beyond that block. Other monitors then automatically move to the next block. Because the pointer moves before verification, no two readers consume the same block.

Role of Metadata

The simple movement of Start/End pointers is not enough on its own – metadata flags (BEING_WRITTEN, BEING_READ) are essential for correctness (See cases in Reading & Writing Two methods – Metadata).

- **BEING_WRITTEN:** indicates the next block is incomplete. In no-overlap mode the monitor denies/queues the read and leaves Start unchanged, preventing half-read errors. In overlap mode, it may begin reading while pacing to the writer.
- **BEING_READ:** indicates the block is still being consumed. In no-overlap mode the camera denies/queues the write. If override is enabled, it raises OVERRIDE and follows the junk-block procedure.

Thus, the pointers provide ordering, while the metadata provides state safety. Together they allow multiple writers and readers to operate in parallel without conflict.
If several cameras and monitors operate in parallel, the pointers guarantee that each writer gets a unique region and each reader gets a unique block in order. The buffer behaves like a shared pipeline – everyone moves forward, no one steps back.

Note: it is possible to add even overlapping in this multi-agent method.

Why it works well

This mechanism works well since the system naturally maintains three invariants:

1. **Uniqueness (claim before use):** pointers move monotonically forward when an agent begins an operation, so no two writers can claim the same region and no two readers can claim the same block. This mechanism guarantees that every block is consumed once, in order, without complex coordination. Each agent knows what is already "owned" by someone else just by looking at the pointers. The system scal**es to many agents while keeping the logic minimal.**
2. **Visibility (state safety):** Per-block metadata exposes in-flight states: BEING_WRITTEN prevents half-reads, and BEING_READ prevents silent overwrites (or forces explicit override). Pointers give order, and metadata gives readiness.
3. **Progress (forward-only pipeline):** Even under override, agents either complete their operation or deny/queue and retry later, no pointer ever moves backward, so the system never desynchronize.

Correctness Perspective

A complete formal proof of correctness would require covering many detailed cases and would take several pages. Instead of presenting the full proof, we highlight a few representative scenarios that demonstrate how the invariants are preserved in practice. These examples make the underlying logic visible without the full formal machinery.

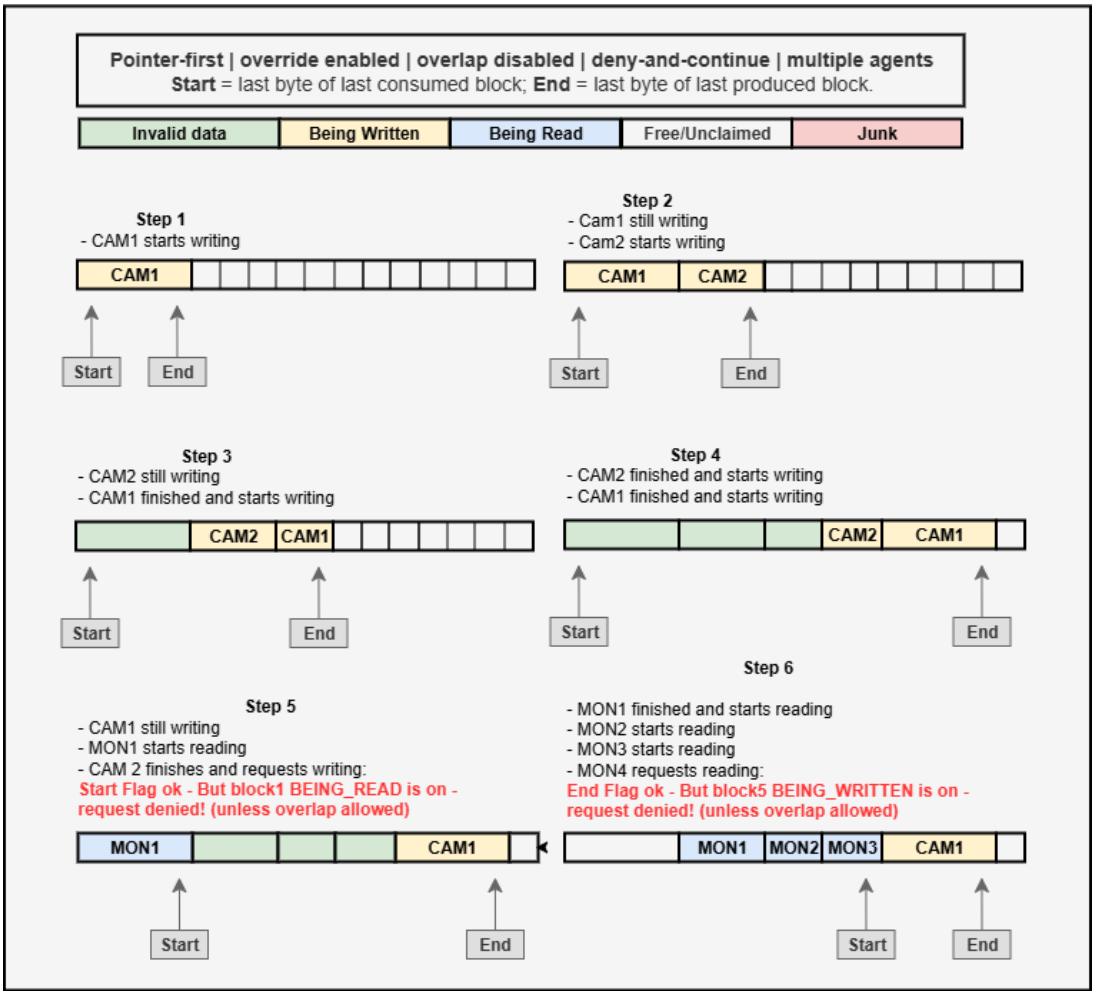


Figure 2: Scaling Example

Further Topics for Exploration (Preview)

Memory Size Estimation

Overview

The main objective is to design the buffer with minimal possible size, since larger buffers increase hardware cost, memory footprint, and access latency. At the same time, the system must control the probability of undesirable states: a full buffer, where no new data can be written (without overriding), or an empty buffer, where the monitor has nothing to read. To address this, flow-control and policy decisions are required – both to prevent these situations when possible, and to define how the system should behave when they occur.

Calculation Concept

The buffer size estimation depends on several parameters: the total buffer size, the current occupancy of the buffer at a given starting point, the combined write rate of all camera agents, and the combined read rate of all monitors. From these values, we calculate the net balance between incoming and outgoing rates. This allows us to estimate whether the buffer tends toward growth (risk of becoming full) or depletion (risk of becoming empty), the probability of reaching these worst-case states over time, and the minimal buffer size required to keep these risks within acceptable limits.

In practice, additional factors must also be considered, such as Retry Policy, override behavior, read-while-write configurations, error-handling strategies, and the expected rate of failures for both write and read operations. These implementation details influence both the effective data rates and the probability of buffer saturation or underflow.

In practice, the calculation involves data from many different sources and configuration settings (such as Retry Policy, override behavior, read-while-write configurations, error-handling strategies, and the expected rate of failures for both write and read operations). The more of these parameters and estimates are known in advance, the higher the probability that the buffer will remain stable. As a safeguard, extra spare capacity can always be provisioned, and flow-control mechanisms can be added to further reduce the risk of saturation or underflow.

Examples for Solutions if buffer tends to FULL (input > output)

- **Backpressure:** temporarily deny/queue photo requests.
- **Quality throttle:** auto-switch to smaller frames (binning/GRAY) until below threshold.
- **Rate limit:** cap camera request rate per agent (or total).

Examples for Solutions if buffer tends to EMPTY (output > input)

- **Backpressure:** temporarily deny/queue image requests.
- **Rate limit:** cap monitor request rate per agent (or total).

Binning

Overview

Binning reduces image resolution by combining groups of neighboring pixels into a single sample. This lowers data size and speeds up transfers, at the cost of detail. In our system, binning directly affects photo block size, write throughput, and buffer pressure.

Examples for Binning Algorithms:

- **Average (mean) binning:** take the arithmetic mean of pixels in the group. Produces smooth, blended results and reduces random noise.
- **Weighted average:** give higher weight to the central pixel(s) or brighter pixels. Preserves more edge detail and center sharpness.
- **Max/Min binning:** store only the maximum (brightest) or minimum (darkest) pixel in the group. Preserves highlights (max) or shadows (min).
- **Median binning:** take the median pixel value. More robust to outliers than mean.
- **Hybrid binning:** use mean for luminance (brightness) and median for chroma (color). Balances smoothness with color stability.

Image Compression

Overview

Image compression reduces the size of photo blocks before they are written to memory, lowering bandwidth and storage usage at the cost of extra computation. It requires additional logic in the camera (to compress during capture) and in the monitor (to decompress before display).

Lossless Compression

Lossless compression reduces the block size while preserving every bit of the original data. When decompressed, the photo is identical to the original. Examples: PNG-style, Huffman coding, run-length encoding.

Lossy Compression

Lossy compression achieves much stronger size reduction by discarding detail the algorithm considers less important. When decompressed, the photo is not identical – some fidelity is lost. This is conceptually similar to binning in our system: fewer pixels or simplified color values, smaller size, but reduced detail.

Pros

- Smaller block size – less buffer pressure and faster transfers.
- Can extend effective memory capacity without changing hardware.
- May allow higher frame rates or resolutions under the same constraints.

Cons

- Extra processing overhead on both camera and monitor. Compression and decompression add latency.
- Lossy methods reduce image quality – lossless methods may not compress enough.
- Increased complexity in implementation and error handling.

Examples of simple Lossless Compression Algorithms:

- **Run-Length Encoding (RLE):**
  1. Idea: if the same value repeats many times, store it once together with the count.
  2. Example**:** $\mathrm{AAAAABBBBCCCCDD} \rightarrow (\mathrm{A}, 5), (\mathrm{B}, 4)(\mathrm{C}, 4)(\mathrm{D}, 2)$.
  3. When decoded, you get back exactly the original sequence – no information lost.
- **Huffman Coding:**
  1. Idea: assign shorter binary codes to frequent values, and longer codes to rare values.
  2. Suppose pixel values A appears often, while Z is rare. Instead of storing both with 8-bit codes, we might encode A = 0, B = 10, Z = 11111.
  3. When decoded, every pixel value is exactly restored.

  Works especially well for images with biased color distributions (e.g., large gray areas with only a few bright pixels).

Error Detection Code (EDC)

Overview

Error detection ensures that corrupted data blocks can be identified before they are used. In our system, errors may occur during capture, processing, transfer, or storage, so each block carries information that helps the monitor verify integrity. In our project, we use CRC-32 as the main error-detection method, since it reliably catches a wide range of error patterns with a small overhead.

Pros

- Protects against undetected corruption.
- Simple methods can be very lightweight.
- Stronger codes detect a wide range of error patterns.

Cons

- Adds overhead (extra bits or bytes per block).
- Requires extra computation on both sides.
- Detects errors but does not correct them (unless combined with ECC).

Example for simple EDCs

- **Parity bit:**
  1. Idea: the simplest error-detecting code. One extra bit is added to each block (or byte/word) to indicate whether the number of 1-bits is even or odd
  2. Example**:** $1011 \text{ (has 3 odd)} \rightarrow 10111$
  3. Detection: if a single bit flips, parity no longer matches.

  detects only odd numbers of bit errors (two flipped bits can pass undetected).

- **Checksum:**
  1. Idea: a simple arithmetic code that sums all data words and stores the result (often modulo some number)
  2. Example**:** $\text{sums } [100, 200, 50] = 350 \text{ (case modulu 256)} = 94 \text{ transmit data} + 94$.
  3. Detection: catches many random errors, especially single-bit and small bursts.

Error Detecting Code in Focus: Cyclic Redundancy Check (CRC-32)

- **Sender side:**
  1. Append 32 zero bits.
  2. Divide this value by the fixed CRC-32 generator polynomial (a 32 degree polynomial, usually a standard constant).
  3. Take the 32-bit remainder as the CRC.
  4. Attach the CRC to the original data and send it.
- **Receiver side:**
  1. Receive data + CRC.
  2. Divide the whole block by the same generator polynomial.
  3. If remainder = 0 then no error.
  4. If remainder ≠ 0 then error detected.

  Note: In our system, the CRC is calculated while data is being written (by the IPU) and verified while being read (by the monitor).

Capabilities:

- CRC-32 guarantees detection of any single-bit error, any double-bit error, any odd number of bit errors, and any burst error of up to 32 bits.
- For longer bursts or more complex error patterns, the chance of missing an error is small (about 1 in 2^{32}).

Note: This is the main idea of CRC-32. For full understanding, further study of polynomial selection, the division process, and implementation techniques is required.

Error Correcting Code (ECC)

Overview

While error-detecting codes (EDCs) can only signal that corruption occurred, Error Correcting Codes go further: they can automatically repair certain errors without retransmission. Each block carries redundant information that lets the receiver reconstruct the original data even when some bits are wrong.

Pros

- Can continue operation even with errors.
- No need to retransmit corrupted data.

Cons

- More redundancy overhead than EDCs.
- Higher computational cost.
- Still limited: can only correct up to a certain number of errors.

Example for simple ECC

- **Receiver side:**
  1. Idea: Simplest possible ECC: transmit each bit multiple times (e.g. three times).
  2. Example: $101 \rightarrow 111000111$.
  3. Receiver decides the “true” bit by majority vote.

  Detects and, with high probability, corrects single errors in each group – but at the cost of very high overhead (200% in triple).