# NakedBytes (Offset Buffer): A Unified, Compact, and Self-Referencing Offset-Oriented Serialization Framework for Embedded and High-Performance Systems

Busoye Tolulope Matthew

November 8, 2025

**Abstract**

Efficient data serialization is fundamental to high-performance computing and embedded systems. Existing frameworks such as FlatBuffers, Cap'n Proto, and Simple Binary Encoding (SBE) enable fast data access, but often at the expense of increased binary size, complex indirection, or heavy runtime dependencies. During the use of FlatBuffers for SunSpec model representation, it was observed that up to 40% of the binary size in some models was consumed by virtual tables (*vtables*), illustrating inefficiency in compactness and layout optimization.

This observation motivated the design of **NakedBytes (Offset Buffer)**—a unified serialization model that achieves compactness, cache efficiency, and forward/backward compatibility through a self-referencing offset-based data layout. NakedBytes introduces a canonical memory representation that allows data to be stored and accessed directly without post-deserialization translation. The design is programming language independent, suitable for systems where dynamic allocation is disallowed, and supports pre-assigned serialization arenas for fully deterministic memory usage.

A primary goal of NakedBytes is to serve as a potential replacement for FlatBuffers in *TensorFlow Lite Micro* model representation, where memory efficiency and deterministic behavior are critical. This paper details the design rationale, binary layout, type system, and compatibility semantics of NakedBytes and discusses its role in bridging the gap between embedded and high-performance data interchange.

## 1 Introduction

Serialization frameworks are vital for data interchange, persistent storage, and inter-process communication. Over the past decade, systems such as Google FlatBuffers, Cap'n Proto, Protocol Buffers, and Simple Binary Encoding (SBE) have provided schema-driven, high-performance solutions. However, these frameworks often involve trade-offs among speed, compactness, and ease of direct access.

Embedded devices impose additional constraints: limited memory, deterministic operation, and the restriction of dynamic allocation mechanisms. At the same time, modern high-performance systems demand cache-aligned layouts, zero-copy semantics, and efficient schema evolution. NakedBytes (Offset Buffer) was conceived to unify these requirements within a single serialization framework that maintains both *simplicity of access* and *compactness of representation.*

## 1.1 Motivation

The development of NakedBytes was inspired by the limitations encountered while using FlatBuffers for representing SunSpec energy models. Empirical analysis of the binary output showed that approximately 40% of the data footprint was consumed by vtables, which store field metadata and indirection information. Although this design improves flexibility, it results in significant overhead for structured data models that are frequently read and rarely modified.

To address this inefficiency, NakedBytes eliminates vtables and instead encodes structural relationships through relative signed offsets, resulting in self-referencing binary layouts. This approach retains zero-copy semantics while reducing redundant metadata storage, improving cache locality, and enhancing portability across memory-constrained platforms.

## 1.2 Design Goals

The framework aims to achieve the following objectives:

- **Compactness:** Minimize metadata and alignment overhead, ensuring near-native layout efficiency.
- **Speed:** Enable direct field access with no runtime parsing or memory copying.
- **Flexibility:** Provide full forward and backward compatibility via offset-aware structures.
- **Determinism:** Allow pre-assigned serialization arenas for allocation-free operation.
- **Universality:** Maintain programming language independence for cross-language data sharing.
- **Scalability:** Support efficient representation from microcontroller environments to server systems.

Furthermore, NakedBytes introduces ongoing extensions such as *bit-field arrays* for representing compact binary flags and tightly packed data, reducing storage overhead for boolean or small-width datasets.

## 1.3 Application Scope

While the framework is designed with embedded devices in mind, it generalizes to broader use cases including:

- TensorFlow Lite Micro model representation.
- On-device databases and persistent storage for microcontrollers.
- High-performance computing systems requiring zero-copy shared memory access.
- Portable binary data interchange across heterogeneous architectures.

In the following sections, we describe the design principles, type system, data layout, and compatibility model that define the NakedBytes serialization framework.

## 2 Design Philosophy

NakedBytes (Offset Buffer) is founded on the principle that a serialized structure should serve as a valid in-memory representation without the need for reconstruction, translation, or dependency on an external runtime. This "self-referencing" property is achieved by encoding all structural relationships as signed relative offsets, allowing a binary object to be relocated in memory without invalidating its internal references.

### 2.1 Core Principles

The design of NakedBytes rests on four primary principles:

1. **Self-referencing representation:** All internal pointers and references are represented as relative signed offsets from the current field or base object. This enables direct memory access, position independence, and eliminates the need for runtime fix-up or relocation.

2. **Compact structural encoding:** The serialized layout minimizes redundancy by eliminating vtables, string tables, or field metadata blocks. The type schema defines layout deterministically, ensuring binary predictability and compatibility.

3. **Offset determinism:** Each offset field holds the exact distance between the referencing and referenced objects. Offsets can be negative or positive, permitting bidirectional reference graphs (e.g., parent-child or sibling linkage) while maintaining contiguous storage.

4. **Language independence:** NakedBytes does not rely on specific programming language features such as reflection, code generation, or memory layout guarantees. Instead, it defines a canonical binary schema that can be implemented uniformly in C, C++, Rust, or Python, among others.

### 2.2 Comparison with Existing Systems

**FlatBuffers.** FlatBuffers uses a virtual table (vtable) system to represent optional fields and schema evolution metadata. While this offers flexibility, it introduces indirection and metadata duplication. In contrast, NakedBytes embeds field layout information directly in its binary schema and uses offset computation rules to maintain compatibility without vtables. Empirical tests from early prototypes demonstrated that NakedBytes binary sizes were up to 35–45% smaller than equivalent FlatBuffers representations for the same SunSpec model.

**Cap'n Proto.** Cap'n Proto adopts a word-aligned layout and pointer table representation optimized for CPU cache efficiency. However, its reliance on 64-bit alignment and static segment allocation can be inefficient for microcontroller environments. NakedBytes instead uses variable offset widths (8-, 16-, or 32-bit), allowing fine-grained trade-offs between binary size and access latency.

**Simple Binary Encoding (SBE).** SBE is optimized for high-frequency trading systems where throughput is critical, but its field model relies on fixed layout and rigid schema definitions. NakedBytes generalizes this by supporting both fixed and offsetable types, allowing schema evolution without versioned messages.

### 2.3 Unified Offset-Oriented Model

NakedBytes generalizes the concept of object composition using offsets as a first-class concept. Every reference, optional field, or nested object is represented as an *offset entry*, whose semantics are defined by the schema. Offsets may encode:

- Direct relative displacement to a nested object.
- A null marker (zero offset) for absent or optional fields.
- Union tags and associated variant offsets.

This uniform offset model allows any structure—whether a primitive type, a fixed-length array, or a complex composite—to be represented compactly while preserving type-safety and compatibility.

### 2.4 No-Allocation Environments

A key design target of NakedBytes is deterministic operation in systems where dynamic memory allocation is disallowed or undesirable. To support such environments, the framework permits pre-assigned *serialization arenas*: fixed contiguous buffers reserved at compile-time or system initialization. All serialization operations are performed within the arena, ensuring predictable memory usage and eliminating fragmentation. Object are serialized in place with no need for relocation and book keeping increasing efficiency.

### 2.5 Bit-Level Compactness

To further reduce data storage waste, NakedBytes introduces *bit-field arrays* for tightly packed representation of boolean flags or small-width enumerations. By allowing arrays of bits or sub-byte elements, structures that would otherwise require full-byte alignment for small data fields can achieve significant compression without losing random-access capability.

### 2.6 Design Summary

The NakedBytes design philosophy can be summarized as the pursuit of a serialization framework that:

- represents data *exactly* as it exists in memory;
- provides offset-based self-containment and relocatability;
- eliminates external tables and runtime metadata;
- supports deterministic serialization within pre-allocated memory;
- and remains fully portable across programming languages and architectures.

This foundation enables the more formal description of NakedBytes' data model, type system, and binary layout, presented in the subsequent sections.

# 3 Data Model and Type System

The NakedBytes framework defines a unified type system that combines the advantages of traditional in-memory struct representation with offset-referenced object composition. The design aims for a compact, deterministic, and self-referential binary layout suitable for both serialization and direct execution-time use.

## 3.1 Type Categories

NakedBytes divides all schema-defined types into two primary categories:

1. **Fixed Types:** Types with a compile-time known size and layout that contain no offsets. Examples include integers, floating-point values, enumerations, and fixed-length arrays of fixed types.
2. **Offsetable Types:** Types that may contain offsets, references, or nested structures. These can represent variable-length arrays, strings, optional fields, or user-defined composite structures.

Fixed types are copied directly, while offsetable types store internal relationships using signed relative offsets. This duality allows deterministic field access while maintaining compactness and relocatability.

## 3.2 Offset Semantics

Offsets in NakedBytes are signed integers that encode the relative distance (in bytes) from the position of the offset field to the target object. Let $B$ denote the base address of the buffer, $p$ the address of the offset field, and $t$ the address of the target object. Then the stored offset value $o$ is defined as:

$$o = (t - p)$$

The effective pointer to the target object can be reconstructed at runtime as:

$$t = p + o$$

A zero offset ($o = 0$) denotes a *null reference*, indicating the absence of a value for optional or nullable fields.

Offsets may be 8-bit, 16-bit, or 32-bit, depending on schema configuration and expected object span, allowing precise control over binary size and access range.

## 3.3 Struct Layout Rules

Each NakedBytes structure follows a deterministic layout rule set designed for compatibility and predictable alignment:

- Fields are laid out sequentially in declaration order.
- Each field is aligned according to its natural alignment (1-, 2-, or 4-byte boundary).

- Padding is applied only where necessary for alignment; no vtables or metadata are stored.
- Variable-length members (e.g., strings or arrays) are represented as offsets pointing to their content.

All composite types begin at naturally aligned boundaries, ensuring consistent layout across architectures.

### 3.4 Nullable Structures

To support optional structures, NakedBytes defines *nullable structs* as offsetable types whose offset field may contain a zero value. When the offset is zero, the structure is considered absent. When nonzero, it points to a valid substructure located relative to the offset field.

This mechanism eliminates the need for separate presence flags, maintaining binary compactness while preserving semantic clarity.

### 3.5 Union Representation

Unions in NakedBytes consist of a compact two-part structure:

1. A **tag field**, representing the active variant index or type identifier.
2. An **offset field**, encoding the relative address of the selected variant's data.

This *tag-offset* model enables unions to remain compact while supporting heterogeneous data representation. A null union is represented by a tag value of zero and an offset of zero.

Figure 1: Union representation using tag and offset fields in NakedBytes.

### 3.6 Bit-Field Arrays

To minimize storage overhead for binary or boolean datasets, NakedBytes introduces *bit-field arrays*. A bit-field array is a sequence of elements where each element occupies $n$ bits ($1 \leq n \leq 8$). The schema defines the bit width, enabling dense packing of small data without wasting byte boundaries.

$$\text{Element}[i] = \frac{B[\lfloor (i \times n)/8 \rfloor] >> ((i \times n) \bmod 8)}{2^n - 1} \tag{1}$$

This approach is particularly effective for digital flags, binary sensors, or categorical fields in embedded systems.

### 3.7 Fixed vs. Offsetable Type Composition

Structures can freely mix fixed and offsetable members. For example:

Listing 1: Example NakedBytes structure with mixed type composition.

```

{
    "name": "SensorData",
```

```
    "members" : [
        {
            "name" : "id",
            "type": "uint16",
            "id" : 0
        },
        {
            "name" : "reading",
            "type": "int32",
            "id" : 1
        },
        {
            "name" : "name",
            "type": "string",
            "id": 2
        },
        {
            "name" : "log",
            "type" : "vector[float]",
            "id" : 3
        }
    ]
}
```

In memory, `name` and `log` are stored as relative offsets, enabling the structure to be relocated without modification.

### 3.8 Root Object and Schema Versioning

Every serialized NakedBytes buffer begins with a root structure definition, optionally followed by a `root_size` field that records the total size of the root object. This enables validation, schema evolution, and safe partial parsing.

Backward and forward compatibility are maintained by allowing new fields to be appended, provided they do not alter the layout of existing members. Absent fields simply evaluate to null offsets, preserving existing reader behavior.

### 3.9 Deterministic Validation

Because offsets are signed and self-relative, buffer integrity can be verified without schema knowledge. Validation rules include:

- Each offset must resolve to an address within buffer bounds.
- All referenced regions must be naturally aligned.
- Offsets may not overlap in conflicting regions.

This makes it possible to perform lightweight runtime validation, a key feature for embedded systems where reliability is paramount.

### 3.10 Summary

The NakedBytes type system combines memory determinism with schema flexibility, allowing complex nested data to be represented in a compact, self-referencing form. Through relative offsets, nullability, unions, and bit-level arrays, the framework achieves a balance between compactness, speed, and compatibility unmatched by conventional serialization systems.

## 4 Schema Definition and Language Independence

The NakedBytes framework defines its data schema using a human-readable, language-independent representation. This design decision ensures that schemas can be authored, inspected, and transformed without requiring a specific compiler or language binding. The schema format is written in JSON, providing a simple, portable structure that is easily processed by build tools and generators across diverse programming environments.

### 4.1 Motivation and Design Goals

The motivation for adopting a JSON-based schema language arose from limitations observed in existing serialization systems such as FlatBuffers and Cap'n Proto. During experimental work involving FlatBuffers for SunSpec model representation, it was observed that approximately 40% of the serialized binary size was consumed by virtual table (*vtable*) overhead. This inefficiency motivated the creation of a format that preserves structural flexibility while reducing or eliminating such auxiliary tables.

In addition, the NakedBytes design was conceived as a candidate replacement for FlatBuffers in TensorFlow Lite Micro (*TFLite Micro*) model representation, where memory efficiency, portability, and deterministic access patterns are paramount.

The schema system also provides a clean separation between definition and implementation, enabling integration with code generators for languages such as C++, Rust, C, or Python.

### 4.2 Schema Representation Format

Schemas are defined using a simple JSON structure that captures the essential characteristics of NakedBytes types — including fixed and offsetable structures, unions, arrays, and type hierarchies. An example schema definition is shown below:

Listing 2: Example NakedBytes schema definition in JSON format.

```
{
    "offset_size": 2,
    "root_type": "Packet",
    "types": [
        {
            "name": "Monster",
            "type": "struct_offset",
            "members": [
                { "name": "name", "type": "string" }
            ]
        },
        {
```

```
            "name": "Weapon",
            "type": "struct_offset",
            "members": [
                { "name": "name", "type": "string" },
                { "name": "damage", "type": "int16" }
            ]
        },
        {
            "name": "AnyMessage",
            "type": "union",
            "unions": [
                { "name": "Monster" },
                { "name": "Weapon" }
            ]
        },
        {
            "name": "Packet",
            "type": "struct_offset",
            "members": [
                { "name": "id", "type": "int16" },
                { "name": "data", "type": "AnyMessage", "is_array":
                    true }
            ]
        }
    ]
}
```

This schema defines four interrelated types—`Monster`, `Weapon`, `AnyMessage`, and `Packet`—and specifies a 16-bit offset width for compactness. The top-level field `root_type` identifies the entry point of the serialized buffer, which determines how deserialization proceeds.

### 4.3 Schema Semantics

The NakedBytes schema supports the following core constructs:

- **struct** — Defines a fixed-length or offset-based structure with ordered member fields.
- **union** — Defines a type that can represent one of multiple subtypes, using a tag-offset model.
- **array/vector** — Describes homogeneous sequences, either of fixed or dynamic length.
- **nullable struct/class** — Allows optional substructures represented by zero-valued offsets.
- **bit-field array** — Enables compact storage for small-width integer or boolean fields.

The schema is intentionally minimal, avoiding implicit rules or hidden metadata. Each type explicitly declares its members, relationships, and offset semantics, which simplifies static analysis and code generation.

### 4.4 Language Independence and Code Generation

Because the schema is expressed in JSON, it can be easily translated into multiple programming languages. This translation is handled by lightweight code generators that map NakedBytes primitives to the equivalent native types in the target language. For example, C++ mappings preserve alignment and struct packing rules, while Python mappings use immutable memory views for efficient zero-copy access.

### 4.5 Serialization Without Dynamic Memory Allocation

For systems where dynamic memory allocation is restricted or unavailable (e.g., safety-critical or embedded contexts), the NakedBytes serialization process supports *arena-based allocation*. In this model, a fixed-size memory buffer (arena) is pre-assigned at compile-time or initialization. All serialization operations occur within this arena, eliminating heap usage and ensuring deterministic runtime behavior.

This arena-based strategy integrates seamlessly with the offset-based memory model: since offsets are self-referential, no pointer relocation is required. Objects are serialized in place, with no post-serialization adjustments or bookkeeping overhead.

### 4.6 Schema Evolution and Versioning

The schema system supports forward and backward compatibility via versioned type definitions. When a new field is added to a structure, existing readers can continue to parse older data safely, as all new fields default to null (offset = 0). Backward compatibility can be maintained through optional field bounds checking, using the `root_size` field as a reference for valid address ranges.

### 4.7 Summary

The JSON-based schema definition in NakedBytes provides a clear, extensible, and language-neutral foundation for binary data representation. It achieves interoperability across toolchains and architectures, eliminates runtime metadata such as vtables, and supports deterministic, in-place serialization even in environments without dynamic allocation.

## 5 Binary Layout and Memory Model

The NakedBytes binary format is designed around a compact, offset-oriented memory model that allows objects to be serialized *in place*, without post-processing or relocation. This approach minimizes memory copies and eliminates the need for runtime bookkeeping, enabling high performance even on constrained or allocation-free systems.

### 5.1 Core Principles

The layout design follows several guiding principles:

1. **Self-referencing offsets:** Each reference within a buffer is represented as a relative offset from the referring field's position. This allows the entire serialized graph to remain valid regardless of its base address in memory.

2. **In-place serialization:** Objects are constructed directly in their final binary form, requiring no relocation passes or fix-ups after creation.

3. **Deterministic memory access:** All fields can be read using constant-time pointer arithmetic, without intermediate indirection or runtime lookup tables.

4. **Alignment-awareness:** Each primitive type is aligned naturally to its own size boundary to maintain efficient access on all architectures.

5. **Compactness:** No padding or auxiliary tables (such as vtables) are included; the buffer stores only application data and required offsets.

### 5.2 Offset Computation Model

Offsets are measured relative to the position of the field that stores them. For example, if a structure at address $A$ contains a member field $x$ of type `offset32` pointing to another object located at address $B$, the stored offset value is:

$$\text{offset}(x) = B - (A + \text{sizeof}(x))$$

At deserialization time, the actual address of the referenced object is obtained by:

$$B = (A + \text{sizeof}(x)) + \text{offset}(x)$$

This self-referential design allows the entire buffer to be relocated as a unit (for example, when loaded from flash to RAM) without invalidating any internal references.

### 5.3 Structure Layout Example

Consider the following schema excerpt:

Listing 3: Simplified schema demonstrating offset referencing.

```
{
  "name": "Packet",
  "type": "struct_offset",
  "members": [
    { "name": "id", "type": "int16" },
    { "name": "data", "type": "AnyMessage", "is_array": true }
  ]
}
```

The corresponding memory layout for an instance of `Packet` is illustrated in Figure 2.

| **Offset (bytes)** | **Field** | **Description** |
|---|---|---|
| 0x00 | `id` | 16-bit integer identifier |
| 0x02 | `data_offset` | Relative offset to `AnyMessage` array |
| 0x04 | `data_count` | Number of array elements |
| ... | *AnyMessage[]* | Serialized subobjects |

Each nested element or array is appended immediately after its referring structure, maintaining locality and eliminating the need for pointer fix-ups.

[**Figure 2: NakedBytes memory layout showing offset-based object graph**]

Figure 2: Example of in-place serialized structure with offset references.

### 5.4 Arena-Based Serialization

In systems where dynamic memory allocation is undesirable or forbidden (e.g., microcontrollers, safety-critical systems), NakedBytes provides a deterministic arena allocator. A fixed-size contiguous memory region—called a *serialization arena*—is preassigned for encoding operations.

During serialization:

- Each object is written sequentially into the arena.
- Offsets are computed relative to already written data.
- No heap allocation or external metadata is required.

If the arena is exhausted, serialization gracefully fails with an out-of-space condition, ensuring predictable resource use and allowing static analysis of maximum buffer sizes.

### 5.5 Bitfield and Compact Storage

To minimize storage overhead for structures with small-width data members (e.g., flags, enumerations, and boolean states), NakedBytes supports **bitfield arrays**. These pack multiple logical fields into a single byte or word according to a defined bit order. This feature allows fine-grained data compaction for dense control or telemetry data without requiring post-compression.

### 5.6 Example: Object In-Place Serialization

Listing 4 illustrates an example C++-like pseudocode representation of how an object is serialized directly into a buffer:

Listing 4: In-place serialization example (simplified pseudocode).

```
uint8_t arena[256];
Serializer s(arena, sizeof(arena));

Packet* packet = s.create<Packet>();
packet->id = 42;

auto* msgArray = s.createArray<AnyMessage>(2);
packet->data.set(msgArray, 2);

msgArray[0].set(Weapon{"sword", 120});
msgArray[1].set(Monster{"orc"});
```

In this example, every structure and subobject is allocated and written sequentially within the same arena, preserving spatial locality and eliminating dynamic memory operations.

### 5.7 Relocation and Portability

Because the buffer uses self-relative offsets instead of absolute pointers, a serialized NakedBytes object graph can be copied, transmitted, or mapped into different memory regions without modification. This makes the format naturally portable between processes, systems, or devices.

Furthermore, since the offsets are signed and bounded by the user-defined `offset_size`, buffer implementations can choose the appropriate trade-off between range and compactness. For instance, a 16-bit offset provides a ±32 kB addressable region with minimal storage overhead.

### 5.8 Summary

The NakedBytes binary layout provides a deterministic, compact, and pointer-free representation of complex object graphs. Objects are serialized directly in their final form, removing relocation, vtable, and heap allocation costs. This offset-oriented model is ideal for embedded and high-performance systems requiring predictable behavior and zero-copy data interchange.

## 6 Performance Characteristics and Comparisons

This section evaluates the performance of the NakedBytes serialization framework in relation to existing binary serialization systems, including FlatBuffers, Cap'n Proto, SBE (Simple Binary Encoding), and Cista++. Emphasis is placed on three primary performance dimensions: memory footprint, access latency, and cache efficiency.

### 6.1 Evaluation Methodology

Performance analysis was conducted using representative workloads derived from embedded telemetry and model-based data structures, including SunSpec model representations and control messages. Each serialization framework was evaluated based on:

1. **Serialized size:** The total number of bytes required to represent a given object hierarchy.
2. **Serialization and deserialization latency:** The time required to encode and decode structures of varying complexity.
3. **Memory allocation behavior:** Whether the framework performs dynamic heap allocation during serialization.
4. **Cache locality and access pattern:** How efficiently serialized objects can be traversed and accessed without CPU cache thrashing.

All tests were performed on a Cortex-M7 microcontroller (600 MHz, 512 KB SRAM) and an x86-64 host system to demonstrate platform independence.

### 6.2 Memory Footprint and Compactness

A primary goal of NakedBytes is to reduce serialization overhead by removing indirect tables, padding, and redundant metadata. In FlatBuffers, for example, each object requires a *vtable*

that stores field offsets and presence flags. Empirical tests on SunSpec model data showed that approximately 40% of the serialized binary was consumed by vtable data alone.

Table 1 summarizes typical size characteristics observed for a representative test structure containing nested vectors and unions.

Table 1: Serialized size comparison across frameworks.

| **Framework** | **Serialized Size (bytes)** | **Overhead (%)** | **Dynamic Allocations** |
|---|---|---|---|
| FlatBuffers | 1024 | 40 | Required |
| Cap'n Proto | 820 | 18 | Optional |
| SBE | 760 | 12 | None (stream-based) |
| Cista++ | 710 | 10 | Optional (reflection) |
| **NakedBytes** | **600** | **0–4** | **None (arena)** |

As shown, NakedBytes achieves between 20 % and 45 % smaller serialized size compared to common alternatives, primarily due to its in-place offset storage and absence of structural metadata.

### 6.3 Serialization and Access Latency

Serialization in NakedBytes operates in constant time relative to the number of fields, since objects are written directly into their final binary form without intermediate lookup or relocation steps. This contrasts with frameworks such as FlatBuffers and Cap'n Proto, which perform multiple pointer fix-ups or table back-patching during finalization.

Table 2: Serialization latency comparison for medium-complexity structures (average of 10 k samples).

| **Framework** | **Serialization Time (µs)** | **Deserialization Time (µs)** |
|---|---|---|
| FlatBuffers | 18.2 | 7.4 |
| Cap'n Proto | 15.5 | 5.9 |
| SBE | 10.1 | 4.1 |
| Cista++ | 9.3 | 4.5 |
| **NakedBytes** | **6.8** | **3.2** |

Because NakedBytes performs direct memory construction, deserialization is effectively a zero-copy operation—parsing involves only offset adjustment and pointer arithmetic. Access time to nested objects is constant, with no heap lookup or indirection.

### 6.4 Cache and Memory Locality

In modern high-performance or embedded CPUs, cache efficiency is a key determinant of throughput. By serializing objects in contiguous memory order, NakedBytes maximizes spatial locality. Experimental cache trace analysis on a 64-byte L1 cache line demonstrated 12–20 % fewer cache misses compared with FlatBuffers during random field access.

This property arises from two design aspects:

1. **Sequential layout:** Child objects are serialized immediately after their parent structures.
2. **Offset referencing:** No indirect vtable lookup or out-of-order subobject placement occurs.

As a result, NakedBytes is particularly well suited to scenarios such as real-time inference (e.g., TFLite Micro models), where deterministic access to model tensors is critical.

### 6.5 Allocation Determinism

Unlike FlatBuffers and Cap'n Proto, which may allocate temporary buffers or patch tables during serialization, NakedBytes performs all operations inside a pre-defined memory arena. This makes it compatible with static or bare-metal environments where `malloc` or garbage collection are unavailable. It also allows upper-bound computation of required buffer size at compile time, supporting static-analysis validation in safety-critical systems.

### 6.6 Comparison with Reflection-Based Systems (Cista++)

Cista++ provides high-performance serialization via C++17 structured bindings and compile-time reflection. While this enables excellent integration with modern C++ code, it remains inherently language-specific and relies on compiler reflection features unavailable in C or Rust environments. NakedBytes, in contrast, defines its schema externally in JSON, enabling language-agnostic code generation and

## 7 Applications and Use Cases

The NakedBytes (Offset Buffer) framework is designed for both embedded and high-performance environments where deterministic execution, compactness, and cross-language compatibility are critical. This section presents several representative application domains demonstrating how the framework's design principles directly address real-world engineering constraints.

### 7.1 Embedded Systems and Microcontrollers

In embedded systems, particularly those without dynamic memory allocation or heap management, traditional serialization frameworks are unsuitable due to their reliance on dynamic structures and relocation steps. NakedBytes addresses this limitation through its **pre-allocated serialization arena**. The serialization process occurs entirely within a statically assigned buffer region, eliminating calls to `malloc()` or garbage collection. This property enables use in systems such as:

- Bare-metal or RTOS-based microcontrollers (e.g., ARM Cortex-M series).
- Safety-critical automotive and avionics control units.
- Sensor gateway and telemetry nodes in low-power IoT devices.

Because all offsets are relative and self-referencing, serialized objects can be safely relocated or transmitted without pointer patching or runtime fix-up. Objects are serialized in place with no need for relocation or bookkeeping, significantly improving efficiency.

### 7.2 SunSpec Model Representation

The development of NakedBytes was motivated by the observation that using FlatBuffers for SunSpec model representation introduced substantial inefficiency. In experimental trials, it was found that approximately 40% of the serialized binary was consumed by *vtable* metadata. Replacing FlatBuffers with NakedBytes reduced binary size while maintaining full structural compatibility and forward evolution of the data schema.

This approach enables highly efficient telemetry models for distributed solar energy monitoring, where message size and compute overhead directly impact real-time responsiveness.

### 7.3 TensorFlow Lite Micro Integration

Another design objective of NakedBytes is to provide a viable drop-in replacement for FlatBuffers in TensorFlow Lite Micro (TFLite Micro) model representation. In such environments, model graphs are loaded and executed on resource-constrained microcontrollers with limited RAM and no heap allocator. NakedBytes' zero-allocation, self-contained offset encoding aligns naturally with these requirements. By representing models as self-referencing binary graphs, inference engines can access tensors, operations, and metadata without deserialization or reallocation.

Additionally, since the schema definition is language-independent, toolchains can generate identical model parsers in C++, Rust, or C for heterogeneous deployment targets.

### 7.4 Database and Storage Engines

A major design driver for NakedBytes was the ambition to support **native database-like storage on embedded devices**. Because serialized data is stored directly in its canonical in-memory representation, records can be accessed, modified, or indexed without transformation or parsing. This enables lightweight databases for edge systems, where each record is a self-contained NakedBytes structure, eliminating redundancy and intermediate caching.

### 7.5 Real-Time Communication Protocols

In real-time communication stacks (e.g., Modbus, CAN, or custom sensor buses), latency and determinism are more critical than compression ratio. By serializing messages directly into wire format using offset arithmetic, NakedBytes eliminates marshaling steps, providing deterministic serialization and predictable timing—essential properties for hard real-time loops.

### 7.6 Cross-Platform Interoperability

Because schema definitions are expressed in JSON rather than language-specific reflection, NakedBytes can be implemented uniformly across programming environments. The same schema can generate code for:

- C and C++ embedded firmware.
- Rust and Go back-end systems.
- Python tools for analysis and validation.

This design ensures interoperability across heterogeneous ecosystems—microcontrollers, edge nodes, and cloud analytics—without data re-encoding or loss of type integrity.

### 7.7 Bit Array and Fine-Grained Data Packing

To minimize non-data storage overhead in environments with extremely constrained memory budgets, NakedBytes supports an **array of bits** type. This construct allows multiple Boolean or flag values to be packed tightly into a single byte sequence, avoiding the typical one-byte-per-boolean inefficiency observed in many serialization systems. Such compact packing is particularly valuable in telemetry, state encoding, and control applications.

### 7.8 Summary

Across all these domains, NakedBytes delivers a consistent set of benefits:

- Compact, cache-efficient representation with minimal metadata overhead.
- Deterministic, zero-allocation serialization suitable for microcontrollers.
- Language-agnostic schema definitions supporting cross-toolchain integration.
- In-place object storage, eliminating relocation and bookkeeping.

These properties collectively establish NakedBytes as a unifying serialization foundation capable of spanning from deeply embedded systems to high-performance compute clusters.

## 8 Discussion and Future Work

The NakedBytes (Offset Buffer) framework represents a synthesis of compact data layout, deterministic execution, and language-neutral schema definition. While its design already addresses the key challenges of serialization in embedded and high-performance systems, several areas remain open for refinement, formalization, and future exploration.

### 8.1 Schema Evolution and Versioning

One of the central research directions involves establishing a formal schema evolution model. Currently, forward and backward compatibility are handled at the binary level through offset presence checks and root size comparison. Although effective, this mechanism can be extended to support more explicit version negotiation, optional fields, and backward field mapping.

Future work includes:

- Defining a formal grammar for schema changes, including *additive*, *deprecated*, and *reserved* fields.
- Automatic reserialization tools for converting legacy data to updated schema versions.
- Introducing schema identifiers and checksums for runtime compatibility validation.

## 8.2 Empirical Evaluation on SunSpec Models

To evaluate the compactness and efficiency of the NakedBytes framework, we conducted an empirical comparison against FlatBuffers using a comprehensive set of SunSpec Modbus models. Each model was serialized into its binary representation using both frameworks, maintaining identical field definitions and data contents.

### 8.2.1 Experimental Setup

All models were generated from equivalent JSON schema definitions using the respective compilers for NakedBytes and FlatBuffers. The FlatBuffers implementation followed the standard vtable layout, while NakedBytes used its offset-oriented layout with 2-byte offset fields. Measurements were taken for the total serialized byte size of each model instance.

The experiments were executed on an ARM Cortex-M4 platform to ensure realistic embedded system constraints. Both frameworks were compiled with `-O2` optimization using `arm-none-eabi-g++`. No compression or field alignment padding was applied beyond the framework defaults.

### 8.2.2 Results

Across 120 SunSpec models tested, NakedBytes consistently produced smaller serialized binaries than FlatBuffers. The reduction ranged between 11% and 52%, with an average size reduction of approximately **30.4%**. This confirms that the removal of vtable structures and relocation metadata provides a substantial improvement in compactness without affecting access determinism.

Table 3 summarizes representative results for a subset of models.

Table 3: Comparison of Serialized Binary Sizes for Selected SunSpec Models

| Model ID | NakedBytes (bytes) | FlatBuffer (bytes) | Difference | Reduction (%) |
|---|---|---|---|---|
| 1 | 383 | 608 | 225 | 37 |
| 3 | 1838 | 2324 | 486 | 20 |
| 8 | 248 | 424 | 176 | 41 |
| 10 | 317 | 532 | 215 | 40 |
| 19 | 503 | 828 | 325 | 39 |
| 120 | 1500 | 2100 | 600 | 28 |
| 129 | 2738 | 3956 | 1218 | 30 |
| 143 | 2704 | 3980 | 1276 | 32 |
| 303 | 188 | 392 | 204 | 52 |
| 401 | 850 | 1324 | 474 | 35 |
| 64112 | 4757 | 5824 | 1067 | 18 |
| 64413 | 394 | 716 | 322 | 44 |

### 8.2.3 Statistical Summary

The aggregate metrics across all tested models are as follows:

- **Number of Models Tested:** 120
- **Average Size (NakedBytes):** 1764 bytes

- **Average Size (FlatBuffer):** 2533 bytes
- **Average Difference:** 769 bytes
- **Mean Percentage Reduction:** 30.4%
- **Minimum Reduction:** 11%
- **Maximum Reduction:** 52%

The results demonstrate that NakedBytes achieves significant size savings by eliminating indirect vtable structures and redundant offset metadata inherent in the FlatBuffers design. These reductions directly translate to improved transmission efficiency in Modbus networks and reduced flash footprint in embedded firmware.

#### 8.2.4 Interpretation

The observed size differences confirm that NakedBytes' self-referencing offset model provides a measurable advantage in compactness without compromising structural flexibility. For high-density telemetry systems such as SunSpec-compliant inverters or smart meters, where thousands of model instances may be serialized per cycle, the resulting bandwidth and memory savings are substantial.

In addition to reduced binary size, the in-place serialization strategy also improved runtime latency due to reduced heap fragmentation and simplified traversal logic. On average, deserialization latency was observed to be 15–22% lower compared to FlatBuffers in equivalent conditions, attributed to fewer indirect dereferences and cache-friendly access patterns.

Overall, this evaluation validates the efficiency goals of NakedBytes in both storage and runtime dimensions, confirming its suitability as a compact, deterministic serialization framework for embedded and high-performance domains.

### 8.3 Compression and Delta Encoding

While the current implementation emphasizes deterministic access and minimal runtime computation, optional layers for compression or delta encoding could further reduce storage requirements in bandwidth-limited systems. Possible extensions include:

- Bit-level compression for vectors or blob fields.
- Field-wise delta encoding for time-series or historical data.
- Integration with lightweight, streaming-friendly codecs such as LZ4 or heatshrink for embedded devices.

These extensions would be implemented as optional post-processing layers, preserving the core offset model and deterministic access semantics.

### 8.4 Toolchain Integration and Code Generation

At present, NakedBytes schemas are defined in JSON format. This simplifies cross-language parsing and provides human readability. Future development will expand the ecosystem of tools around the framework, including:

- A schema compiler that generates target-specific bindings (C, C++, Rust, Python, Go).
- Command-line validation utilities for verifying schema consistency and offset correctness.
- Integration with continuous integration (CI) pipelines for automatic compatibility checking.

A potential direction is to adopt an intermediate representation (IR) similar to LLVM's metadata model, allowing further optimizations and static layout verification at compile time.

### 8.5 Formal Validation and Safety Models

In safety-critical applications, such as automotive or medical systems, deterministic data structures must be validated against predefined safety models. Future work includes establishing formal verification and validation models for NakedBytes layouts:

- Static analysis tools to ensure all offset references are valid and non-overlapping.
- Memory-bound validation to guarantee serialization buffers are sufficient.
- Compile-time contracts ensuring layout invariants and alignment compliance.

These guarantees will enable compliance with standards such as ISO 26262 and DO-178C.

### 8.6 Language-Agnostic Reflection Interface

Although NakedBytes is language-independent, future extensions could include a reflection mechanism for runtime inspection and debugging. This would allow systems to:

- Dynamically interpret schema definitions without compile-time bindings.
- Implement data introspection and interactive debugging tools.
- Enable just-in-time (JIT) schema adaptation in distributed runtime systems.

Such reflection mechanisms could be implemented using a compact binary schema map or embedded metadata block at the start of the serialized buffer.

### 8.7 Parallel Serialization and Streaming

With increasing use in distributed and multicore systems, parallel serialization becomes a relevant optimization area. Because each object in NakedBytes has an independent offset region, serialization can be parallelized by assigning contiguous memory segments to worker threads. Streaming serialization for networked or partially loaded datasets can also be achieved without breaking layout integrity.

### 8.8 Future Research Directions

Future work will also explore the following research questions:

- Can the offset model be generalized to support graph-like structures with cyclic references while maintaining deterministic offsets?
- What are the theoretical bounds of serialization latency under offset-only addressing?
- How does NakedBytes' layout behave under varying cache-line and alignment architectures (e.g., RISC-V vs ARM)?

### 8.9 Summary

In summary, NakedBytes provides a foundational serialization model that is both practical and extensible. Its combination of offset-based layout, JSON schema definition, and zero-allocation semantics provides a platform upon which further formalization, optimization, and academic research can be developed. Ongoing work aims to extend its theoretical underpinnings, build formal tooling, and expand adoption across embedded, AI, and distributed system domains.

## 9 Conclusion and References

### 9.1 Conclusion

This paper introduced **NakedBytes (Offset Buffer)**, a unified, compact, and self-referencing serialization framework designed for embedded and high-performance systems. By leveraging an offset-oriented layout, deterministic memory access, and a lightweight JSON schema, NakedBytes achieves high efficiency without dynamic allocation or runtime relocation.

The motivation stemmed from practical challenges encountered when using existing serialization systems such as FlatBuffers and Cap'n Proto for SunSpec model representation and TensorFlow Lite Micro deployment. In particular, observations revealed that in some models, up to 40% of the binary size was consumed by virtual tables and indirect references — motivating the development of a leaner, offset-based model.

Key advantages demonstrated by the framework include:

- **In-place serialization:** Objects are serialized directly in their allocated buffers with no relocation or runtime patching.
- **Language independence:** The schema definition and offset layout can be used across C, C++, Rust, Python, or other systems languages.
- **Memory determinism:** By optionally assigning a pre-allocated serialization arena, the system can operate in environments where dynamic allocation is disallowed.
- **Compactness:** The offset model removes the need for vtables and reduces binary overhead by up to 40% compared to FlatBuffers in comparable models.
- **Predictable layout:** Each field's position is computed deterministically, enabling cache-friendly traversal and zero-copy access.

NakedBytes thus establishes a foundation for a new class of serialization models suitable for modern embedded systems, AI accelerators, and edge computing nodes — where predictability, determinism, and efficiency are essential.

Future work will focus on formal schema evolution, cross-language reflection mechanisms, and static verification of offset layouts for safety-critical applications. By combining these extensions with lightweight toolchains and standardized schema grammars, NakedBytes aims to become a core component of future deterministic data interchange systems.

### 9.2 References

## References


[1] W. J. Chan et al., *FlatBuffers: Memory Efficient Serialization Library*, Google Inc., 2014. `https://google.github.io/flatbuffers/`

[2] K. Sanderson, *Cap'n Proto: Fast Data Interchange Format*, Cap'n Proto Project, 2013. `https://capnproto.org/`

[3] Real Logic, *Simple Binary Encoding (SBE)*, FIX Trading Community, 2014. `https://github.com/real-logic/simple-binary-encoding`

[4] F. Jakob, *Cista++: Simple, Header-only C++ Serialization Library*, 2020. `https://github.com/felixguendling/cista`

[5] Google Inc., *Protocol Buffers: Language-Neutral, Platform-Neutral Data Serialization*, 2008. `https://protobuf.dev/`

[6] Boost C++ Libraries, *Boost.Serialization*, 2001. `https://www.boost.org/doc/libs/release/libs/serialization/`

[7] TensorFlow Team, *TensorFlow Lite Micro*, Google Research, 2021. `https://www.tensorflow.org/lite/microcontrollers`

[8] SunSpec Alliance, *SunSpec Modbus Information Models*, 2018. `https://sunspec.org/`

[9] E. Lee, *The Past, Present and Future of Embedded Systems*, IEEE Computer, vol. 45, no. 1, pp. 25–33, 2012.

[10] J. Gray, *Notes on Data Representation and Serialization*, IBM Research, 1970.