Experimental Evaluation of CANopen Communication on ESP32 Master–Slave Nodes

Kenneth Buah

November 18, 2025

Abstract

Controller Area Network (CAN) is widely used in embedded systems for robust, bus-based communication. CANopen is a higher-layer protocol built on top of classical CAN 2.0 that introduces a standardized Object Dictionary, network management, and communication services such as PDO, SDO, SYNC, and Heartbeat. This report documents an experimental attempt to implement a minimal CANopen master–slave setup on two ESP32-based nodes. The objectives were to understand the basic CANopen architecture, configure a simple Object Dictionary, and establish visible CANopen communication between the nodes. The work focuses on the practical steps and difficulties encountered, in particular the challenge of observing CANopen traffic over a serial interface despite having messages defined in the Object Dictionary. The results show that while CANopen provides a rich and structured communication model, its configuration, tooling, and integration overhead can be significant for small, time-constrained prototype setups.

1 Introduction

Embedded systems frequently require multiple microcontroller-based nodes to exchange data reliably over a shared bus. Controller Area Network (CAN) is a common solution for this problem due to its robustness, arbitration mechanism, and wide adoption in automotive and industrial domains. However, CAN 2.0 itself does not define how data in frames should be interpreted or how devices should be configured and managed.

CANopen addresses this by defining a higher-layer protocol on top of CAN 2.0. It introduces a standardized Object Dictionary (OD), communication objects, and a network management state machine. In principle, this structure simplifies interoperability and reuse, especially when devices from different vendors are involved.

The purpose of this project was to explore, in a practical way, how difficult it is to get a basic CANopen communication scenario running on a general-purpose microcontroller platform. Specifically, the goals were:

- to implement a minimal CANopen master–slave setup on two ESP32 boards;
- to configure an Object Dictionary with a small set of application variables;
- to exchange these variables using CANopen mechanisms (PDO and/or SDO);
- to observe and verify CANopen communication, for example via serial output logs.

In practice, although entries were defined in the Object Dictionary, it was initially not possible to see any CANopen-related communication on the serial console. This report describes the background, implementation steps, and the reasons why this issue occurred, as well as general observations on using CANopen in this context.

2 Background

2.1 CAN 2.0

Classical CAN 2.0 defines a multi-master serial bus where nodes communicate by sending frames that contain:

- an identifier (11-bit or 29-bit);
- up to 8 bytes of data;
- control bits for arbitration, error handling and acknowledgement.

The protocol itself is agnostic to the meaning of the identifiers and payload. Designers are free to assign identifiers to messages and define their payload formats as needed. This freedom is powerful but can lead to non-standardized solutions.

2.2 CANopen

CANopen is a higher-layer protocol specified by CiA (CAN in Automation). It provides a standardized way to build systems on top of CAN 2.0 through:

- an Object Dictionary (OD) that models all device data (parameters, I/O, status);
- a set of communication objects (PDO, SDO, SYNC, NMT, Heartbeat, EMCY);
- a defined network management (NMT) state machine;
- a convention for mapping OD entries into CAN messages.

Each device is assigned a *node ID* in the range 1–127. CANopen then derives the CAN IDs (COB-IDs) for different message types by adding offsets to the node ID, which leads to a predictable mapping between node identity and CAN frames.

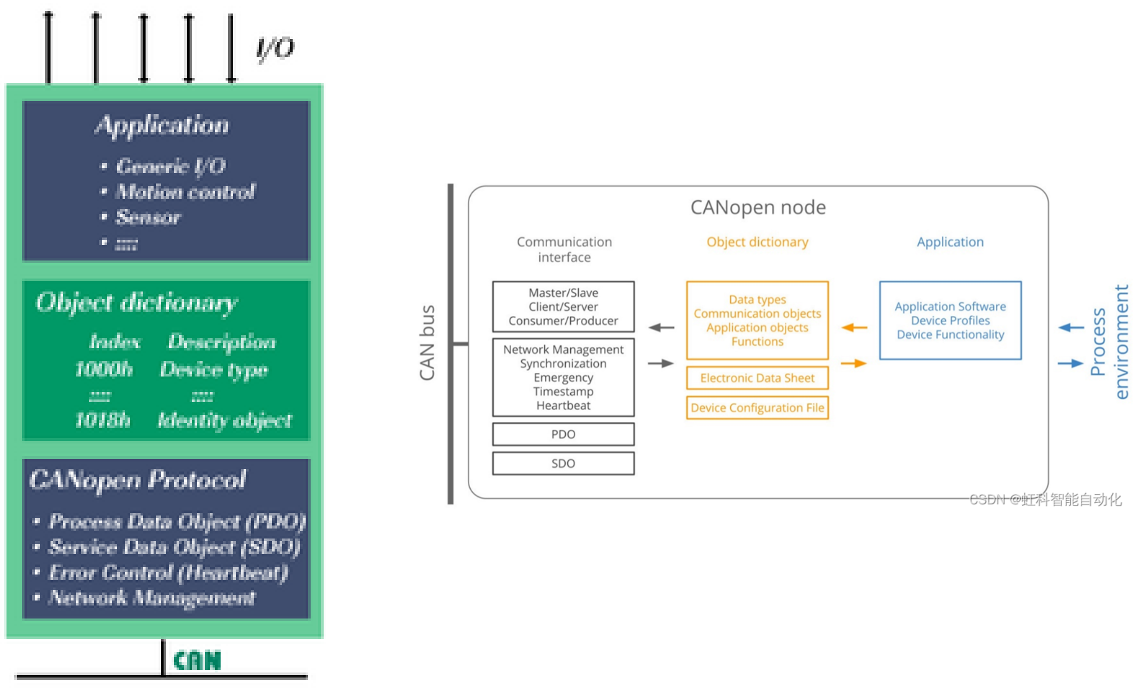
I/O
Application
• Generic I/O
• Motion control
• Sensor
• ::::
Object dictionary
Index Description
1000h Device type
:::: ::::
1018h Identity object
CANopen Protocol
• Process Data Object (PDO)
• Service Data Object (SDO)
• Error Control (Heartbeat)
• Network Management
CAN
CANopen node
CAN bus
Communication interface
Master/Slave Client/Server Consumer/Producer
Network Management Synchronization Emergency Timestamp Heartbeat
PDO
SDO
Object dictionary
Data types Communication objects Application objects Functions
Electronic Data Sheet
Device Configuration File
Application
Application Software Device Profiles Device Functionality
Process environment
CSDN @虹科智能自动化

Figure 1: Illustrative views of the CANopen architecture. The Object Dictionary sits between the CANopen protocol services (PDO, SDO, NMT, Heartbeat, etc.) and the application layer.

2.3 Object Dictionary

The Object Dictionary is a per-device table of entries, each identified by:

- an *index* (16-bit, e.g. `0x2000`);
- a *sub-index* (8-bit, e.g. `0x01`).

Each OD entry has a data type, access rights, and a description. Some indices are standardized:

- `0x1000` — Device type;
- `0x1017` — Producer heartbeat time;
- `0x1800`.../`0x1A00`... — TxPDO communication and mapping;
- `0x1400`.../`0x1600`... — RxPDO communication and mapping;
- `0x2000`... — manufacturer-specific entries.

In this project, the `0x2000` range is used for simple demonstration variables.

2.4 Communication Objects

The key CANopen communication objects relevant for this work are:

- **PDO (Process Data Object)**: carries real-time data in a single CAN frame, typically mapped directly to a set of OD entries.
- **SDO (Service Data Object)**: used for confirmed read/write access to arbitrary OD entries via a client–server mechanism.
- **SYNC**: a broadcast message that acts as a timebase for synchronous PDOs.
- **NMT (Network Management)**: commands to change node states (Initialization, Pre-operational, Operational, Stopped).
- **Heartbeat / Boot-up**: messages through which nodes announce their presence and current NMT state.

3 System Setup and Example Application Model

The experimental system consisted of:

- two ESP32 boards, each connected to a CAN transceiver;
- a CAN bus with proper termination resistors;
- one node configured as a CANopen master;
- one node configured as a CANopen slave.

The goal of the demo was intentionally simple: the master should be able to write a small variable to the slave (for example, a control value), and the slave should periodically publish status information back to the master using PDOs.

3.1 Example Object Dictionary Entries

To support this, the following manufacturer-specific OD entries in the `0x2000` range were used on the slave:

- `0x2000:01` — `uint16` control value (e.g. a setpoint);
- `0x2000:02` — `uint16` measured value or feedback;
- `0x2000:03` — `uint8` status flag (0 = OK, non-zero = error).

These entries were then mapped into PDOs as follows:

- **RxPDO1** (`0x200 + nodeID`) — carries the control value (`0x2000:01`);
- **TxPDO1** (`0x180 + nodeID`) — carries the measured value and status flag.

The master uses an SDO client or a configured PDO to update `0x2000:01` on the slave, while the slave periodically transmits TxPDO1 with `0x2000:02` and `0x2000:03`.

4 Implementation on ESP32

4.1 Software Architecture

The implementation used the open-source `ESP32_CanOpenNode` project as a starting point [1]. The architecture relied on several FreeRTOS tasks:

- a CAN control task to initialize and manage the ESP32 CAN (TWAI) driver;
- a main CANopen task to run the stack state machine and time-based processing;
- optional tasks to print CAN traffic to the serial console.

A simplified structure is shown in Listing 1.

Listing 1: High-level structure of the CANopen application on ESP32

```
void canopen_app_init(void) {
  // 1) Initialize serial for debugging
  Serial.begin(115200);

  // 2) Bring CAN (TWAI) up, set nodeID, bit-rate, filters
  // 3) Initialize CANopen stack and Object Dictionary
  // 4) Configure heartbeat time, SYNC period, PDO mappings
  // 5) Enter PRE-OPERATIONAL, finish setup, then go to OPERATIONAL
}

void canopen_app_process(void) {
  // Periodic processing:
  // - Handle timers (SYNC, heartbeat)
  // - Process SDO client/server
  // - Produce and consume PDOs
  // - Update application variables mapped in the OD
}

void can_rx_tx_interrupt(void) {
  // CAN receive/transmit interrupt:
  // - Read frames from RX FIFO and pass them to the CANopen stack
  // - Trigger transmission of pending frames
  // Keep this function short; heavy work happens in canopen_app_process()
}
```

The main loop periodically calls `canopen_app_process()`, while the CAN interrupt handler ensures that incoming frames are handled with minimal latency.

4.2 Bring-Up Procedure

The bring-up sequence for the two-node system was:

1. Choose node IDs and bit-rate (e.g. node 1 as master, node 2 as slave).
2. Configure the CAN controller and filters on both ESP32 boards.
3. Initialize the CANopen stack and Object Dictionary on each node.

4. Configure Heartbeat producer and consumer times using OD entries such as `0x1017`.
5. Define PDO mappings for RxPDO1 and TxPDO1 to link CAN frames with OD entries.
6. Start the CANopen main task and transition nodes to Operational.

4.3 Serial Output and Message Logging

An important aspect of this experiment was verifying that CANopen communication actually occurred. The initial expectation was that, once OD entries and PDOs were configured, CANopen-related messages would automatically appear on the serial console.

In practice, this did not happen. Although the Object Dictionary contained entries and the CANopen stack was running, there was no visible CANopen traffic on the serial monitor. The reason is that the CANopen stack processes frames internally but does not automatically print them. Serial output depends on explicit logging code, for example:

- a dedicated task that reads CAN frames from a queue and prints them;
- logging macros or functions that send formatted messages to `Serial`.

In the example code from the repository, such a print task existed but was initially disabled (commented out), and logging was performed via macros that only produce output if the debug level is configured appropriately. As a result, even though messages were defined in the OD and the CAN interface was active, no CANopen communication was visible on the serial console until the logging components were explicitly enabled.

5 Experimental Observations and Challenges

5.1 Object Dictionary Versus Observable Output

One key observation was that having a well-defined Object Dictionary and PDO configuration does not guarantee observable output on the serial interface. The CANopen stack operates on CAN frames and OD entries internally; unless additional code is present to log these operations, the serial monitor remains silent.

This led to an initial misconception: it seemed as if "CANopen communication was not working" because nothing appeared on the serial console, even though the underlying stack might have been exchanging frames on the bus. The experiment highlighted that:

- the Object Dictionary controls how data is structured and exchanged on the CAN bus;
- the serial console is a separate debugging channel that requires explicit logging.

5.2 Configuration Effort

Implementing even a simple master–slave CANopen demo involved:

- defining OD entries and ensuring consistent data types;
- mapping PDOs correctly with bit-accurate lengths and offsets;
- configuring heartbeat times and node states;
- setting up SDO client and server roles as needed.

For a small two-node prototype, a significant part of the time was spent understanding and debugging the configuration, rather than developing application-specific logic.

5.3 Tooling and Debugging

Another challenge was the lack of specialized CANopen-aware tools in the development environment. While generic CAN analyzers can display raw CAN frames, interpreting them in terms of CANopen semantics (e.g. identifying PDOs, SDO sequences, and NMT transitions) is more difficult without protocol decoding support.

On the serial side, the absence of default logging meant that even when OD entries were defined and communication objects were configured, the system appeared inactive. Only after enabling or implementing explicit serial logging could the internal CANopen activity be observed and analysed.

6 Conclusion

This report presented an experimental evaluation of CANopen communication between two ESP32-based nodes in a master–slave configuration. The primary goal was to understand how CANopen structures data and messages through the Object Dictionary and communication objects, and to establish a simple demonstration in which the master writes a control variable to the slave and the slave reports status back.

The work showed that CANopen offers a rich, standardized model for device communication and management, but also that this richness comes with non-trivial configuration and debugging overhead. In particular:

- Defining OD entries and PDO mappings is necessary but not sufficient to see any output on the serial console; explicit logging must be added to visualise CANopen traffic.
- For a small, time-constrained prototype with only two nodes, a substantial portion of effort is spent on understanding and configuring the protocol rather than on application-level functionality.

Although the experiments did not result in a fully successful and stable demonstration of CANopen communication between the two ESP32 nodes, they provided valuable insights into how CANopen operates and what is required to integrate it on a microcontroller platform such as the ESP32. Future work could focus on:

- using a CANopen-aware analyser to decode and display frames directly on the bus;
- building dedicated logging and visualisation tools for educational demonstrations;
- comparing CANopen with simpler, custom CAN-based protocols in terms of implementation effort versus benefits for small embedded networks.

References

1. T. Haanstad, *ESP32_CanOpenNode*, GitHub repository, available at https://github.com/thaanstad/ESP32_CanOpenNode/tree/main (accessed 2025).