CubeSat Space Protocol (CSP)

Introduction and Beginner's Guide

Written and prepared by:
Andreas Chr. Dyhrberg, ufelectronics.eu

Version: 2025.10.04 - 1500

Introduction

CubeSat Space Protocol (CSP) is a **lightweight network protocol** designed for small spacecraft such as CubeSats. Think of CSP as the "**Internet inside a satellite**" — a way for all the electronic subsystems to talk to each other efficiently and predictably.

CSP was created at **Aalborg University (Denmark)** to solve a common problem: every student satellite team was inventing its own internal communication protocol. CSP standardized it — simple, efficient, and reliable, even for small microcontrollers.

Why CSP Exists

A CubeSat is like a tiny distributed computer system. Inside, you often have:

- An **On-Board Computer (OBC)** that runs mission software.
- An **Electrical Power System (EPS)** that measures battery levels and manages solar input.
- A **Communications board (COMMS)** handling radio links.
- A **Payload** (e.g., a sensor, camera, or experiment).

Each needs to send and receive data to others:

- "EPS to OBC: Battery = 7.3 V."
- "OBC to COMMS: Transmit housekeeping data."
- "Payload to OBC: Science data ready."

CSP provides the rules and structure for these exchanges — **packets, addresses, routing, and delivery** — in a small, easy-to-implement form suitable for microcontrollers.

How CSP Works (Simplified)

CSP is modeled after the Internet Protocol (IP) but optimized for space.

- Every subsystem is a **node** with its own address.
- Each packet has a **header** (source, destination, priority, etc.) and a **payload** (the data).
- Packets travel across standard hardware buses: **UART**, **CAN**, **I²C**, or **Ethernet**.

CSP includes layers similar to the Internet model:

1. **Physical layer** – how data moves (UART, CAN, etc.)
2. **Data link** – framing and basic error checking
3. **Network** – addressing and routing between nodes
4. **Transport** – optional reliability (like TCP vs UDP)

Because it's modular, CSP can run on an 8-bit AVR microcontroller or a full Linux computer — the same architecture, just scaled.

What You Can Do With CSP

CSP enables:

- **Subsystem communication**: The OBC polls the EPS for voltage or temperature data.
- **Command and telemetry**: The ground station sends commands, which are routed to the right subsystem.
- **Data transfers**: Science data or logs are sent as CSP packets between nodes.
- **Simulation and testing**: Teams can run CSP networks on the ground before building real hardware.

What Makes CSP Special

- **Designed for space:** built for low power, low memory, and reliability.
- **Cross-platform:** runs on Linux, FreeRTOS, and bare-metal systems.
- **Open source:** freely available under the LGPL license.
- **Space-proven:** used in multiple real CubeSat missions.
- **Modular:** works over many types of physical connections.

Official repository: https://github.com/libcsp/libcsp

What libcsp Writes About itself, alias CSP

Source: https://github.com/libcsp/libcsp and https://libcsp.github.io/libcsp/

Cubesat Space Protocol (CSP) is a small protocol stack written in C. CSP is designed to ease communication between distributed embedded systems in smaller networks, such as Cubesats. The design follows the TCP/IP model and includes a transport protocol, a routing protocol and several MAC-layer interfaces. The core of `libcsp` includes a router, a connection oriented socket API and message/connection pools.

The protocol is based on an very lightweight header containing both transport and network-layer information. Its implementation is designed for, but not limited to, embedded systems with very limited CPU and memory resources. The implementation is written in GNU C and is currently ported to run on FreeRTOS, Zephyr and Linux (POSIX).

The idea is to give sub-system developers of cubesats the same features of a TCP/IP stack, but without adding the huge overhead of the IP header. The small footprint and simple implementation allows a small 8-bit system to be fully connected on the network. This allows all subsystems to provide their services on the same network level, without any master node required. Using a service oriented architecture has several advantages compared to the traditional master/slave topology used on many cubesats.

- Standardised network protocol: All subsystems can communicate with each other (multi-master)
- Service loose coupling: Services maintain a relationship that minimizes dependencies between subsystems
- Service abstraction: Beyond descriptions in the service contract, services hide logic from the outside world
- Service reusability: Logic is divided into services with the intention of promoting reuse.
- Service autonomy: Services have control over the logic they encapsulate.
- Service Redundancy: Easily add redundant services to the bus
- Reduces single point of failure: The complexity is moved from a single master node to several well defined services on the network

Features

- Thread safe Socket API
- Router task with Quality of Services
- Connection-oriented operation (RFC 908 and 1151).
- Connection-less operation (similar to UDP)
- ICMP-like requests such as ping and buffer status.
- Loopback interface
- Very Small Footprint in regards to code and memory required
- Zero-copy buffer and queue system
- Modular network interface system
- OS abstraction, currently ported to: FreeRTOS, Zephyr, Linux
- Broadcast traffic
- Promiscuous mode

Getting Started at Home

You don’t need a satellite to explore CSP — you can start learning and experimenting right on your desk.

Step 1. Choose Your Hardware

You can practice CSP using inexpensive microcontrollers and development boards:

- **Raspberry Pi (any model)** – easiest for testing CSP on Linux.
- **ESP32 boards** – Wi-Fi capable and compatible with FreeRTOS or bare-metal CSP builds.
- **STM32 Nucleo or Discovery boards** – popular in space hardware prototyping.
- **Arduino-compatible boards (e.g., SAMD21/SAMD51)** – for simple serial/UART CSP tests.

To simulate a multi-node spacecraft, use **two boards connected by UART or CAN bus** — one acts as “OBC,” the other as “EPS.”

Step 2. Install CSP Software

Before diving into real hardware, you should get CSP running locally — on your computer — to understand its behavior and communication flow. There are multiple ways to do this, depending on your operating system and experience level.

2.1. The Ideal Starting Point

The **most flexible and error-free setup** for CSP beginners is:

- Use **Linux** (either native or inside Windows Subsystem for Linux, WSL).
- Build and test the **libcsp** package directly on your computer.
- Use CSP’s built-in tools (`csp_ping`, `csp_term`, `csp_iflist`) to simulate network communication between virtual nodes.

This setup avoids cross-compilation and driver issues, so you can focus on understanding how CSP packets flow before adding real hardware.

2.2. Option A – Linux (Recommended for Learning and Development)

Install dependencies

For Ubuntu/Debian-based systems:

```
sudo apt update
sudo apt install git build-essential cmake python3
```

Clone and build CSP

```
git clone https://github.com/libcsp/libcsp.git
cd libcsp
mkdir build && cd build
cmake ..
make
sudo make install
```

Run a local test

```
./examples/csp_server &
./examples/csp_client
```

You'll see output showing packets being sent and received — confirming that your local virtual network is working.

You can explore:

- `./examples/csp_iflist` → lists active interfaces
- `./examples/csp_ping <node>` → ping a node (similar to network ping)
- `./examples/csp_term` → open a terminal session between nodes

2.3. Option B – Windows Users

There are **two reliable approaches** to running CSP on Windows:

(1) Using Windows Subsystem for Linux (WSL2) — *Recommended*

This gives you a native Linux environment inside Windows.

1. Install WSL2 (if not already done):

```
wsl --install
```

Then restart your computer.

2. Open Ubuntu (from the Start Menu).

3. Inside Ubuntu, follow the same Linux commands from Option A.

You now have a working CSP environment inside Windows without any complicated setup.

(2) Using MinGW or MSYS2

If you prefer staying purely in Windows, you can compile CSP using MinGW:

```
pacman -S git make cmake mingw-w64-x86_64-gcc
git clone https://github.com/libcsp/libcsp.git
cd libcsp
mkdir build && cd build
cmake -G "MinGW Makefiles" ..
make
```

However, some examples that rely on Linux-specific interfaces may not work fully (e.g., socket interfaces).

This setup is fine for experimenting, but less ideal for networking tests.

2.4. Option C – macOS Users

macOS provides a Unix-like environment that works almost like Linux.

Install development tools

```
xcode-select --install
brew install git cmake make
```

Clone and build CSP

```
git clone https://github.com/libcsp/libcsp.git
cd libcsp
mkdir build && cd build
cmake ..
make
sudo make install
```

Run example programs

```
./examples/csp_server &
./examples/csp_client
```

2.5. Option D – Microcontroller Boards (ESP32, STM32, Arduino, etc.)

Once you understand how CSP works on your PC, the next step is running it on embedded boards.

Each board type uses its own build environment:

Board Type	Development Environment	Notes
ESP32	PlatformIO or ESP-IDF	FreeRTOS supported, UART or Wi-Fi transport
STM32 (Nucleo, Discovery)	STM32CubeIDE	Reliable CAN/UART communication
Arduino (SAMD21/SAMD51)	PlatformIO	Works for simple UART CSP examples
Raspberry Pi	Native Linux	Full CSP build works directly

For example, to build CSP for ESP32 in PlatformIO, add to `platformio.ini`:

```
[env:esp32dev]
platform = espressif32
board = esp32dev
framework = espidf
lib_deps = https://github.com/libcsp/libcsp
```

You can then write small programs to send and receive packets between two boards via UART.

2.6. Option E – Virtual CSP Network (No Hardware Needed)

If you just want to *see CSP in action* before using real boards, you can run a virtual two-node simulation on your computer.

```
# Start node A (address 1)
./examples/csp_server -a 1 -i loopback &

# Start node B (address 2)
./examples/csp_client -a 2 -i loopback
```

Packets will be sent between the two “nodes” running on your own computer, allowing you to learn how routing, addressing, and commands work before wiring anything physically.

2.7. Troubleshooting Tips

- If compilation fails on Windows, check whether you are in **WSL** or **MSYS2** — Linux-like paths work best.
- Always verify that your **CMake version ≥ 3.15**.
- If your `examples` folder doesn’t appear, enable examples by running:

```
cmake -DBUILD_EXAMPLES=ON ..

make
```

- On microcontrollers, ensure your UART pins are correct and that both nodes share **GND**.

2.8. Next Steps

Once you have CSP running — either virtually or on two real boards — you can begin experimenting with:

- **`csp_ping`** – test connectivity between nodes
- **`csp_term`** – send text commands between nodes
- **`csp_reboot`** – remotely reset a node (simulated or real)

This local setup is the **foundation for real spacecraft networking**. From here, you can extend to radio-based communication, real-time telemetry, and subsystem integration.

Step 3. Your First Project: “EPS Telemetry Simulation”

Goal

Simulate an On-Board Computer (OBC) talking to an Electrical Power System (EPS).

Setup

- Node 1 (OBC) = Raspberry Pi or ESP32 running CSP.
- Node 2 (EPS) = another board that responds to telemetry requests.

Process

1. Assign each node a unique CSP address (e.g., OBC = 1, EPS = 2).
2. The OBC sends a “Get telemetry” packet every 10 seconds.
3. The EPS replies with simulated data: voltage, current, temperature.
4. Display the data on a serial terminal or log file.

Step 4. Expand Your Setup

Once you understand the basics, you can:

- Add a **COMMS node** that sends packets via radio (LoRa, UHF modules).
- Connect a **payload board** that generates random “experiment” data.
- Build a **ground station simulator** on your PC using libcsp tools.

With just two or three microcontrollers, you can mimic how a real CubeSat network operates.

Resources and References

- Official CSP repository: https://github.com/libcsp/libcsp
- Documentation: https://libcsp.readthedocs.io
- Example CubeSat missions using CSP: **GOMX**, **AAUSAT**, **Delphini-1**
- Tutorials and community projects: search for *“libcsp GitHub examples”* or *“AAUSAT CSP test”*

For further learning, explore:

- **PlatformIO** for microcontroller development
- **STM32CubeIDE** for CAN-based communication
- **ESP-IDF** for FreeRTOS-based CSP networks

What This Knowledge Leads To — Why It Matters

Learning the CubeSat Space Protocol (CSP) is more than just understanding how satellites communicate. It's an introduction to **real-world systems engineering** — where electronics, software, and communication merge into one functioning system.

1. A Gateway to Real Space Systems

CSP represents the same communication logic used in professional space missions. By mastering it, you learn **how spacecraft subsystems interact**, how commands and telemetry are structured, and how reliability is engineered in extreme environments.

This knowledge leads directly into roles or studies involving:

- **On-Board Software Development** – writing logic for small satellites and payload control.
- **Space Systems Integration** – connecting multiple subsystems and testing them together.
- **Mission Operations** – interpreting telemetry, planning uplinks, and troubleshooting in orbit.
- **Hardware-in-the-loop (HIL) simulation** – creating full spacecraft testbeds on the ground.

Understanding CSP means understanding how to *architect a spacecraft* — not just how to code for one.

2. Motivation for Learning CSP

CSP transforms abstract space concepts into something you can touch and test. When you set up two boards and see CSP packets move between them, you're recreating what happens between real spacecraft systems — just on your workbench.

You gain:

- A **clear mental model** of subsystem communication.
- A **hands-on grasp** of how spacecraft handle data, commands, and power coordination.
- The **confidence** to design, debug, and integrate embedded systems that must not fail.

It's not about simulation alone — it's about thinking like an engineer who sends real hardware into space.

3. What This Could Lead To

By building and experimenting with CSP networks, you position yourself for opportunities in:

- **Student or research CubeSat missions** (software, integration, or testing).
- **Aerospace startups** and **small satellite companies**, where open-source avionics are common.
- **Universities and space labs** developing experimental payloads or onboard networks.

- **International collaborations** in small spacecraft technology and space education.

Even small CSP demos can become **portfolio projects** that prove you understand real-time communication and embedded integration — highly valued across technical fields.

4. Where Else This Knowledge Is Useful

CSP principles extend far beyond spaceflight. The same architectural thinking is used in:

- **Internet of Things (IoT)** — smart sensors, industrial monitoring, and automation networks.
- **Robotics** — distributed systems for robots or drones communicating in real time.
- **Automotive systems** — CAN bus communication between vehicle control units.
- **Industrial control systems** — reliable data exchange between PLCs and sensors.
- **Embedded systems development** — microcontroller communication in any product.

Once you've learned CSP, you understand how to design small, distributed systems that exchange data predictably and securely — whether it's in orbit, on Earth, or in a robot.

This makes your knowledge **broadly transferable**, turning satellite learning into a foundation for many engineering careers.

5. Benefits of the DIY CSP Setup

Running CSP on your own boards gives you *more than* a test — it gives you an engineering playground.

It's a low-cost way to:

- Learn embedded networking, packet routing, and fault handling.
- Practice modular design and subsystem testing.
- Build confidence in cross-platform development (Linux ↔ microcontroller).
- Gain a real-world skill directly applicable to satellite projects and IoT/robotics alike.

When you complete your first working CSP network — even with two devices on a desk — you've learned something that scales all the way up to real spacecraft.

In essence, **you've built the foundation of system-level engineering** — a mindset that's as valuable on Earth as it is in orbit.

Key Terms for Beginners

Term	Meaning
CSP	CubeSat Space Protocol – communication standard for small satellites.
Node	Any device participating in the CSP network (e.g., OBC, EPS, payload).
Packet	A unit of data sent between nodes, with addressing and a payload.
Header	The part of a packet that contains routing information.
Payload	The actual data in the packet (e.g., telemetry or a command).
UART	Universal Asynchronous Receiver-Transmitter – serial communication line.
CAN Bus	Controller Area Network – robust bus for multiple devices (used in cars and satellites).
I²C	A simple communication bus for sensors and small devices.
Routing	The process of forwarding packets to the correct destination.
libcsp	The official C-language implementation of the CubeSat Space Protocol.
Ground station	Earth-based system that sends commands and receives data from the satellite.
Telemetry	Status data (voltages, temperatures, mode flags) sent from satellite to Earth.
Command uplink	Instructions from Earth to the spacecraft.
LEO	Low Earth Orbit – typically 200–2000 km above Earth.
OBC	On-Board Computer – the satellite's main controller.
EPS	Electrical Power System – manages power generation and distribution.
COMMS	Communication subsystem – handles radio transmission.
Payload	The mission-specific equipment or experiment.