Copy of tafiti-csp-guide

CubeSat Space Protocol

Welcome

Overview

This document serves as a guide for the CubeSat Space Protocol, and it is intended for Tafiti engineers to get them up to speed with the protocol. For further information such as examples, please visit the CSP GitHub https://github.com/libcsp.

Definitions

Browse through the commonly used definitions and/or refer to them when going through the other sections.

Definitions

Get Started with CSP

A comprehensive walkthrough from installation to compilation.

Deep Dive

Definitions

This page contains the commonly used definitions in this guide.

- **Protocol -** a set of rules and conventions that describe how information is to be exchanged between two entities.
- **Protocol stack -** is an implementation of a computer networking protocol suite or protocol family.
- **Packet -** to messages to move more quickly across a network, each message is broken into small fragments and each fragment sent individually. In networking terms, these pieces of messages are called packets.

- **Router** - A specialized computer that is designed to receive incoming packets on many links and quickly forward the packets on the best outbound link to speed the packet to its destination. It serves two primary functions: managing traffic between these networks by forwarding data packets to their intended IP addresses, and allowing multiple devices to use the same Internet connection.
- **Address -** a unique name or number assigned to a computer that is part of a network. For the Internet Protocol(IP), these are known as **IP addresses**.
- **IP Address** - A globally assigned address that is assigned to a computer so that it can communicate with other computers that have IP addresses and are connected to the Internet. To simplify routing in the core of the Internet IP addresses are broken into Network Numbers and Host Identifiers. An example IP address might be “212.78.1.25”.
- **hop -** A single physical network connection. A packet on the Internet will typically make several “hops” to get from its source computer to its destination.
- **server** - a computer that provides the network resources and provides service to other computers when they request it.
- **client** - the computer running a program that requests the service from a server.
- **client server network -** one on which all available network resources such as files, directories, applications and shared devices, are centrally managed and hosted and then are accessed by the client.
- **offset -** The relative position of a packet within an overall message or stream of data.
- **window size -** The amount of data that the sending computer is allowed to send before waiting for an acknowledgement.
- **Local Area Network** (LAN) - A network covering an area that is limited by the ability for an organization to run wires or the power of a radio transmitter.
- **Wide Area Network** (WAN) - A network that covers longer distances, up to sending data completely around the world. A WAN is generally constructed using communication links owned and managed by a number of different organizations.
- **Routing Tables**: Information maintained by each router that keeps track of which outbound link should be used for each network number.
- **datagram** - is a defined block of data with a specified size and structure. This block enters the transport layer as a single datagram and exits the transport layer in the other end as a single datagram.
- **acknowledgement -** When the receiving computer sends a notification back to the source computer indicating that data has been received.
- **buffering -** Temporarily holding on to data that has been sent or received until the computer is sure the data is no longer needed.
- **listen -** When a server application is started and ready to accept incoming connections from client applications.
- **port -** A way to allow many different server applications to be waiting for incoming connections on a single computer. Each application listens on a different port. Client applications make connections to well-known port numbers to make sure they are talking to the correct server application.
- **flow control -** When a sending computer slows down to make sure that it does not overwhelm either the network or the destination computer. Flow control also causes the sending computer to increase the

speed at which data is sent when it is sure that the network and destination computer can handle the faster data rates.

- **socket -** A software library available in many programming languages that makes creating a network connection and exchanging data nearly as easy as opening and reading a file on your computer.
- **ciphertext** - A scrambled version of a message that cannot be read without knowing the decryption key and technique.
- **decrypt -** The act of transforming a ciphertext message to a plain text message using a secret or key.
- **encrypt** - The act of transforming a plain text message to a ciphertext message using a secret or key.
- **plain text -** A readable message that is about to be encrypted before being sent.
- **private key -** The portion of a key pair that is used to decrypt transmissions.
- **public key -** The portion of a key pair that is used to encrypt transmissions.

packet buffer - memory space set aside for storing packets awaiting transmission over networks or storing packets received over networks. These memory spaces are either located in a network interface card (NIC) or in the computer that holds the card.

References

These are the references used to develop this guide, and are also recommended for reading by each Tafiti engineer looking to use CSP.

Links

1. CSP GitHub https://github.com/libcsp
2. Reliable Data Protocol, initial specification RFC 908 - https://www.rfc-editor.org/info/rfc908
3. Reliable Data Protocol, update RFC 1151 - https://www.rfc-editor.org/info/rfc1151

Books

1. Introduction to Networking: How the Internet Works, by Charles Severance
2.

Guide

Deep Dive

This is intended to be a comprehensive guide for potential CSP users.

First start by reading the introduction.

Introduction to CSP

Next, go over the CSP protocol stack to gain a high-level understanding of what CSP entails

The Protocol Stack

Read on the blocks

Build upon the high-level understanding gained from the previous section.

CSP blocks

See the structure of the library

Library Structure

Go hands-on

Currently only implementation on STM32 has been demonstrated but the AVR implementation is in the works.

STM32 implementation

AVR implementation

Introduction to CSP

The **CubeSat Space Protocol** (CSP) is a small protocol stack 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.

History and License Information

The idea was developed by a group of students from Aalborg University in 2008. In 2009 the main developer started working for GomSpace, and CSP became integrated into the GomSpace products.

The three letter acronym CSP was originally an abbreviation for CAN Space Protocol because the first MAC-layer driver was written for CAN-bus. Now the physical layer has extended to include SpaceWire, I2C and RS232, the name was therefore extended to the more general CubeSat Space Protocol without changing the abbreviation.

CSP was initially under the GNU LGPL (Lesser General Public License) but it is now under the **MIT License**, which **allows**:

- Commercial use
- Modification
- Distribution
- Private use

The MIT license does **not** include:

- Warranty
- Liability

Flight Heritage

Here is a list of some of the known satellites or organisations, that use CSP:

- GomSpace GATOSS GOMX-1
- AAUSAT-3
- EgyCubeSat
- EuroLuna
- NUTS
- Hawaiian Space Flight Laboratory
- GomSpace GOMX-3, GOMX-4 A & B

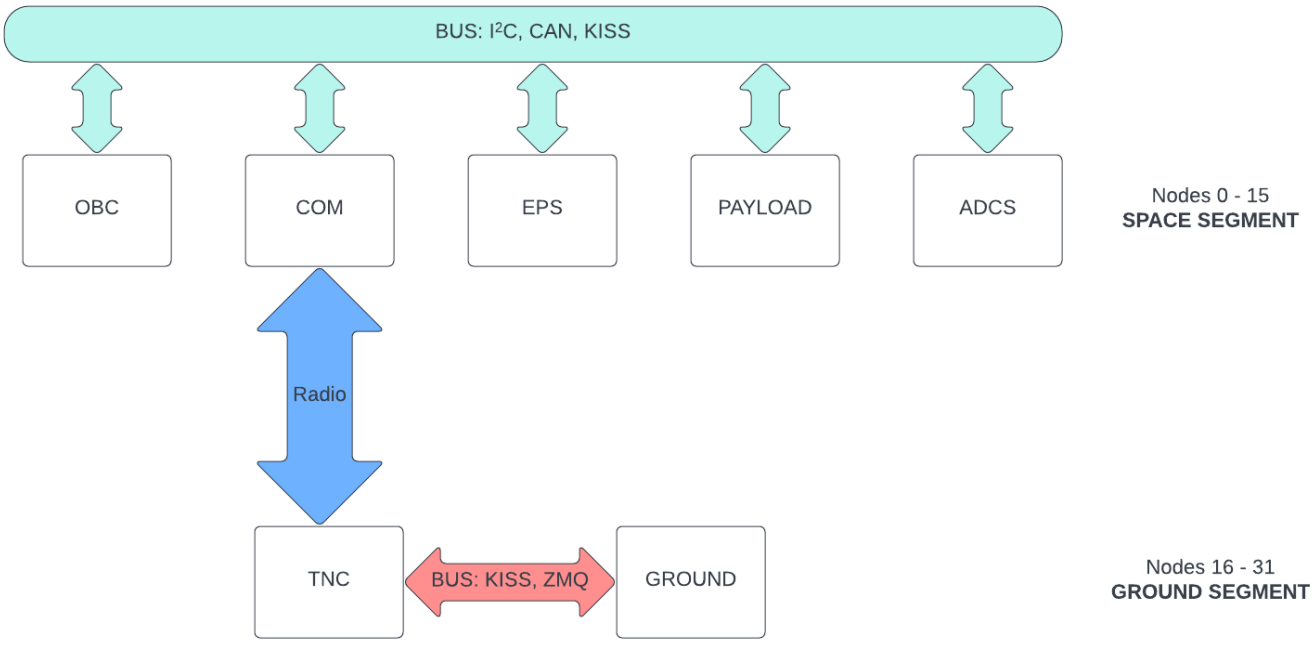
Network Topology

The most common CSP topology has two segments:

- Ground segment
- Space Segment

The address range from 0-31 has been segmented into two equal sized segments:

- Nodes 0-15 (00H - 0FH) - Space Segment
- Nodes 16-31 (10H - 1FH) - Ground Segment



Typical topology of a network using CSP

The basic routing table must be assigned at **compile-time** of each subsystem.

However, each node supports assigning an individual route to every single node in the network and can be changed run-time.

This means that **the network topology can be easily reconfigured after start-up**.

Application of CSP

The protocol is based on a 32-bit header containing both **transport** and **network-layer** information. Its implementation is designed for, but not limited to, embedded systems such as:

- 8-bit AVR microprocessor
- 32-bit ARM and AVR from Atmel

The implementation is written in GNU C and is currently ported to run on **FreeRTOS** or **POSIX** operating systems such as Linux.

The idea is to give sub-system developers of CubeSats the same features of a TCP/IP stack, but without adding the large overhead of the IP header.
The small footprint and simple implementation allows a small 8-bit system with less than 4 kB of RAM 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 mater/slave topology used on many CubeSats.

Why should we use CSP for Tafiti?

- Standardised network protocol: All subsystems can communicate with each other
- 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

The Protocol Stack

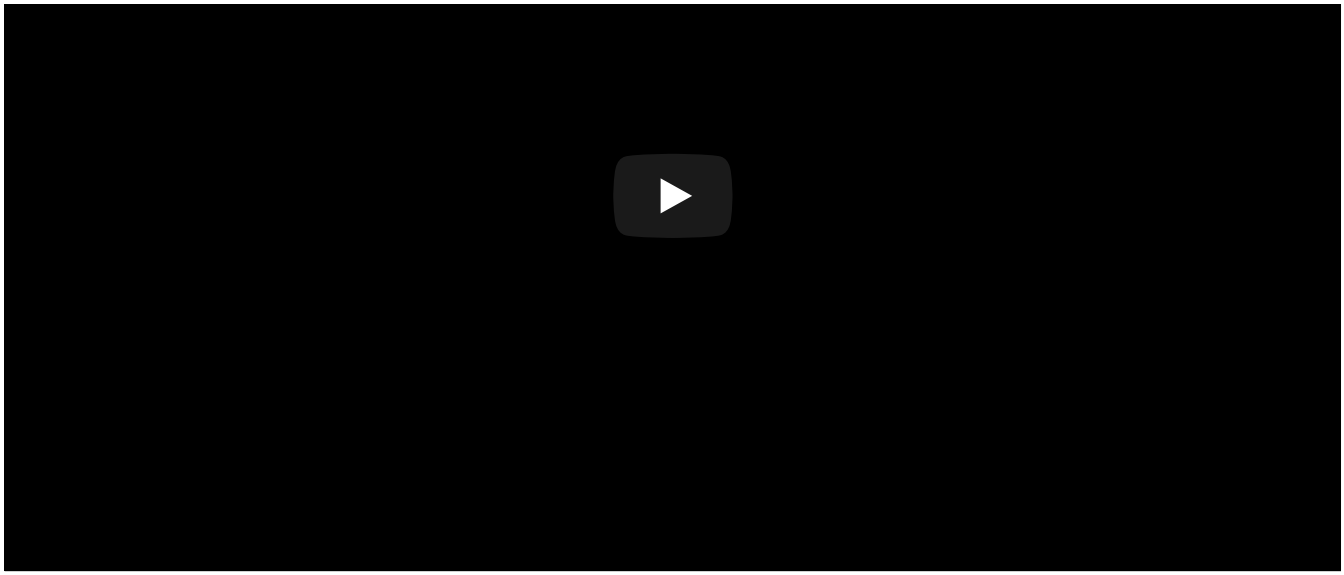
Since CSP is based on the 4 layer TCP/IP model, knowledge of the TCP/IP stack is required, however this tutorial will provide a brief overview.

The 4 layer Transmission Control Protocol and the Internet Protocol (TCP/IP) model

The engineers who built the first internets broke the overall problem into four basic subproblems that could be worked on independently by different groups.

We visualize these different areas as layers stacked on top of each other, with the Link layer on the bottom and the Application layer on the top. The Link layer deals with the wired or wireless connection from your computer to the local area network and the Application layer is what we as end users interact with, such as a browser.

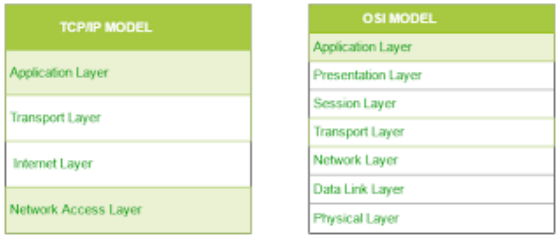
First, see these short YouTube videos to first understand the OSI model and then the TCP/IP model.



The OSI model

The TCP/IP model

TCP/IP layers



TCP/IP layers compared to OSI

1. Link Layer

The Link layer is responsible for connecting your computer to its local network and moving the data across a single hop.

The Link layer needs to solve two basic problems when dealing with these shared local area networks:

- How to encode and send data across the link. If the link is wireless, engineers must agree on which radio frequencies are to be used to transmit data and how the digital data is to be encoded in the radio signal. For wired connections, they must agree on what voltage to use on the wire and how fast to send the bits across the wire eg: RS-485 standard which contains application guidelines, including data signalling rate vs. cable length, stub length, and configurations.
- How to cooperate with other computers that might want to send data at the same time. Techniques like Carrier Sense Multiple Access with Collision Detection (CSMA/CD) ensure fair access to the many different computers connected to the network. When your computer wants to send data, it first listens to see if another computer is already sending data on the network (Carrier Sense). If no other computer is sending data, your computer starts sending its data. But if two computers started sending at about the same time, the data collides, and your computer does not receive its own data. When a collision is detected, both computers stop transmitting, wait a bit, and retry the transmission.

2. **Internetwork Layer**

Once your packet destined for the Internet makes it across the first link, it will be in a router**.** Your packet has a source address and destination address and the router needs to look at the destination address to figure out how to best move your packet towards its destination.

When the packet reaches the last link in its journey, the link layer knows exactly where to send your packet.

3. **Transport Layer (TCP)**

It looks at a packet's destination address and finds a path across multiple network hops to deliver the packet to the destination computer.

But sometimes these packets get lost or badly delayed. Other times the packets arrive at their destination out of order because a later packet found a quicker path through the network than an earlier packet.

As the destination computer reconstructs the message and delivers it to the receiving application, it periodically sends an acknowledgement back to the source computer indicating how much of the message it has received and reconstructed.

4. **Application Layer**

Each application is generally broken into two halves.

- **server -** It runs on the destination computer and waits for incoming networking connections.
- **client -** runs on the source computer.

When we develop the server half and the client half of a networked application, we must also define an application protocol that describes how the two halves of the application will exchange messages over the network.

CSP protocol stack

The CSP protocol stack includes functionality on all layers of the TCP/IP model:

Layer 1: Drivers

CSP is not designed for any specific processor or hardware peripheral, but yet these drivers are required in order to work. The intention of LibCSP is not to provide CAN, I2C or UART drivers for all platforms, however some drivers has only been included for some specific platforms.

If you do not find your driver supported, it is quite simple to add a driver that conforms to the CSP interface. For example, RS-485 drivers specific to CSP have not been written but can be developed.

For good stability and performance **interrupt driven drivers are preferred in favour of polled drivers**. Where applicable also **DMA usage is recommended**.

Layer 2: Media Access Control (MAC) interfaces

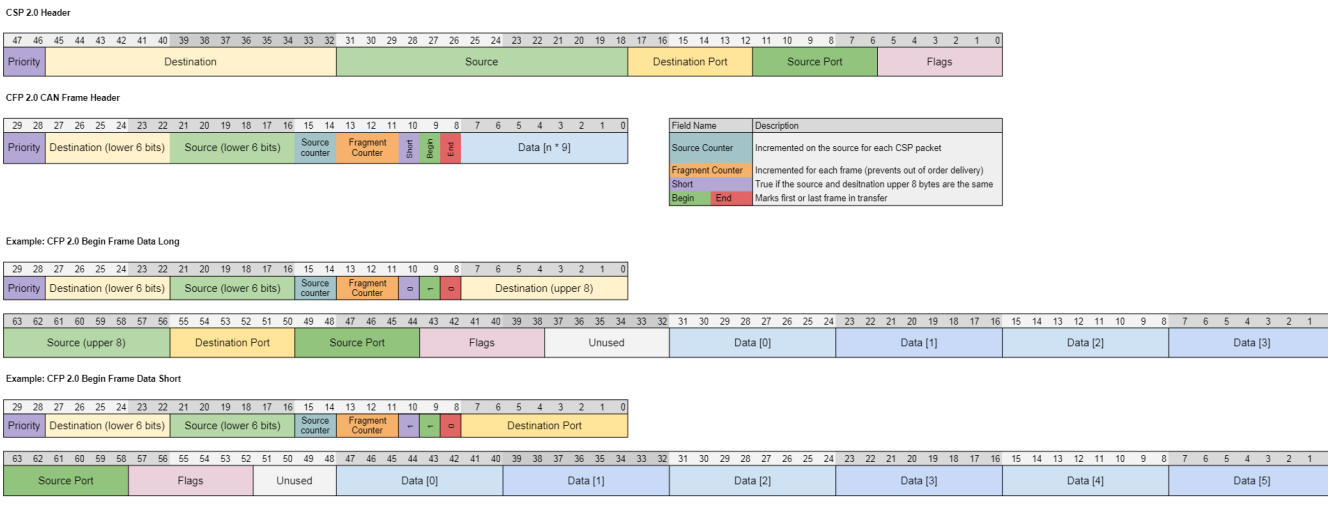
CSP has interfaces for I2C, CAN, RS232 (KISS) and Loopback. The layer 2 protocol software defines a frame-format that is suitable for the media.

CSP can be easily extended with implementations for even more links. For example a radio-link and IP-networks. The file "csp_interface.h" declares the rx and tx functions needed in order to define a network interface in CSP.

During initialisation of CSP each interface will be inserted into a linked list of interfaces that is available to the router.

Layer 3: Network Router

The router core is the backbone of the CSP implementation. The router works by looking at a 32-bit CSP header which contains the destination and source address together with port numbers for the connection.



The CSP header

The router supports both local destination and forwarding to an external destination. Messages will never exit the router on the same interface that they arrives at, this concept is called **split horizon**, and helps prevent routing loops.

The main purpose of the router is to accept incoming packets and deliver them to the right message queue. Therefore, in order to listen on a port-number on the network, a task must create a socket and call the accept() call. This will make the task block and wait for incoming traffic, just like a web-server or similar. When an incoming connection is opened, the task is woken. Depending on the task-priority, the task can even pre-empt another task and start execution immediately.

There is no routing protocol for automatic route discovery, all routing tables are pre-programmed into the subsystems. The table itself contains a separate route to each of the possible 32 nodes in the network and the additional default route. This means that the overall topology must be decided before putting sub-systems together. However CSP has an extension on port zero CMP (CSP management protocol), which allows for over-the-network routing table configuration. This has the advantage that default routes could be changed if for example the primary radio fails, and the secondary should be used instead.

4. Transport Layer

LibCSP implements two different Transport Layer protocols:

- UDP (unreliable datagram protocol)
- RDP (reliable datagram protocol)

The most important thing to notice is that CSP is entirely a datagram service. There is no stream based service like TCP. CSP preserves this datagram structure all the way to the physical layer for I2C, KISS and Loopback interfaces.

Unreliable Datagram Protocol

This is not to be confused with the User Diagram Protocol that is used in TCP/IP.

UDP uses a simple transmission model **without** implicit hand-shaking dialogues for guaranteeing reliability, ordering, or data integrity.

Thus, UDP provides an unreliable service and datagrams may arrive out of order, appear duplicated, or go missing without notice. UDP assumes that error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level.

Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.

Reliable Datagram Protocol

CSP provides a transport layer extension called RDP (reliable datagram protocol) which is an implementation of RFC908 and RFC1151. RDP provides a few additional features:

- Three-way handshake
- Flow Control
- Data-buffering
- Packet re-ordering
- Retransmission
- Windowing
- Extended Acknowledgment

See the relevant references.

CSP blocks

The basics of CSP

The following diagram shows a conceptual overview of the different blocks in CSP. The shown interface is CAN.

```
    buffer                  connection   send   read/accept
       ^                         |         |         ^
       |                         |         v         |
       |   +-----------------+   |     +----------------+    +--------------------+
       |   | connection pool |   |     |    CSP Core    |    | csp_task_router() |
       |   |  csp_conn.c     |<--+---->| * routing      |<---|    csp_route.c    |
       |   +-----------------+         | * crypt        |    +--------------------+
       |                               | * UDP/RDP      |              ^
       |                  csp_route_t  | * CRC32        |              |
       v                 +------------>|                |     +----------------+
+--------------+         |             |                |     | incoming queue |
| buffer pool  |         |             |                |     |     qfifo.c    |
| csp_buffer.c |  +-------------+      +----------------+     +----------------+
+--------------+  |routing table|            |                         ^
       ^          |  rtable.c   |            v                         |
       |          +-------------+      +-------------------------------------+
       |                               |  (next_hop)                         |
```

```
        |                               |    CAN interface (csp_if_can.c)    |
        +------------------------------>|                      csp_can_rx()  |
                                        +-------------------------------------+
                                              |   CAN frame (8 bytes)   ^
                                              v                         |
                                     csp_can_tx_frame()       socketcan_rx_thread()
                                              (drivers/can/can_socketcan.c)
```

Buffer

All buffers are allocated once during initialization of CSP, after this the buffer system is entirely self-contained.

All allocated elements are of the same size, so the buffer size must be chosen to be able to handle the maximum possible packet length.

The buffer pool uses a queue to store pointers to free buffer elements. First of all, this gives a very quick method to get the next free element since the dequeue is an O(1) operation. Furthermore, since the queue is a protected operating system primitive, it can be accessed from both task-context and interrupt-context.

The `csp_buffer_get()` version is for task-context and `csp_buffer_get_isr()` is for interrupt-context. Using fixed size buffer elements that are pre-allocated is again a question of speed and safety.

Definition of a buffer element `csp_packet_t`:

```
/**
   CSP Packet.

   This structure is constructed to fit with all interface and protocols to prevent the
   need to copy data (zero copy).

   @note In most cases a CSP packet cannot be reused in case of send failure, because the
   lower layers may add additional data causing increased length (e.g. CRC32), convert
   the CSP id to different endian (e.g. I2C), etc.
*/
typedef struct {
        uint32_t rdp_quarantine;        // EACK quarantine period
        uint32_t timestamp_tx;          // Time the message was sent
        uint32_t timestamp_rx;          // Time the message was received

        uint16_t length;                        // Data length
        csp_id_t id;                            // CSP id (unpacked version CPU readable)

        uint8_t * frame_begin;
        uint16_t frame_length;

        /* Additional header bytes, to prepend packed data before transmission
         * This must be minimum 6 bytes to accomodate CSP 2.0. But some implementations
         * require much more scratch working area for encryption for example.
```

```
         *
         * Ultimately after csp_id_pack() this area will be filled with the CSP header
         */

        uint8_t header[CSP_PACKET_PADDING_BYTES];

        /**
         * Data part of packet.
         * When using the csp_buffer API, the size of the data part is set by
         * csp_buffer_init(), and can later be accessed by csp_buffer_data_size()
         */
        union {
                /** Access data as uint8_t. */
                uint8_t data[0];
                /** Access data as uint16_t */
                uint16_t data16[0];
                /** Access data as uint32_t */
                uint32_t data32[0];
        };

} csp_packet_t;
```

A basic concept in the buffer system is called **Zero-Copy**. This means that from user space to the kernel-driver, the buffer is never copied from one buffer to another. This is a big deal for a small microprocessor, where a call to `memcpy()` can be very expensive.

This is achieved by a number of `padding` bytes in the buffer, allowing for a header to be prepended at the lower layers without copying the actual payload. This also means that there is a strict contract between the layers, which data can be modified and where.

The padding bytes are used by the I2C interface, where the `csp_packet_t` will be casted to a `csp_i2c_frame_t`, when the interface calls the driver Tx function `csp_i2c_driver_tx_t`:

```
/**
   I2C frame.
   This struct fits on top of a #csp_packet_t, removing the need for copying data.
*/
typedef struct i2c_frame_s {
    //! Not used  (-> csp_packet_t.padding)
    uint8_t padding[3];
    //! Cleared before Tx  (-> csp_packet_t.padding)
    uint8_t retries;
    //! Not used  (-> csp_packet_t.padding)
    uint32_t reserved;
    //! Destination address  (-> csp_packet_t.padding)
    uint8_t dest;
    //! Cleared before Tx  (-> csp_packet_t.padding)
    uint8_t len_rx;
    //! Length of \a data part  (-> csp_packet_t.length)
    uint16_t len;
    //! CSP id + data  (-> csp_packet_t.id)
```

```
    uint8_t data[0];
} csp_i2c_frame_t;
```

Connection

CSP supports both connection-less and connection-oriented connections. See more about protocols in `layer4`.

During initialization libcsp allocates the configured number of connections. The required number of connections depends on the application. Here is a list functions, that will allocate a connection from the connection pool:

> - client connection, call to `csp_connect()`
> - server socket for listening `csp_socket()`
> - server accepting an incmoing connection `csp_accept()`

An applications receive queue is located on the connection and is also allocated once during initialization. The length of the queue is the same for all queues, and specified in the configuration.

Send

The data flow from the application to the driver, can basically be broken down into following steps:

> 1. if using connection-oriented communication, establish a connection> `csp_connect()`, `csp_accept()`
> 2. get packet from the buffer pool: `csp_buffer_get()`
> 3. add payload data to the packet
> 4. send packet, e.g. `csp_send()`, `csp_sendto()`
> 5. CSP looks up the destination route, using the routing table, and calls `nexthop()` on the resolved interface.
> 6. The interface (in this case the CAN interface), splits the packet into a number of CAN frames (8 bytes) and forwards them to the driver.

Receive

The data flow from the driver to the application, can basically be broken down into following steps:

> 1.

1. the driver layer forwards the raw data frames to the interface. in this case CAN frames
2. the interface will aquire a free buffer (e.g. `csp_buffer_get_isr()`) for assembling the CAN frames into a complete packet
3. once the interface has successfully assembled a packet, the packet is queued for routing - primarily to decouple the interface, e.g. if the interfacec/drivers uses interrupt (ISR).
4. the router picks up the packet from the incoming queue and routes it on - this can either to a local destination, or another interface.
5. the application waits for new packets at its Rx queue, by calling `csp_read()` or `csp_accept` in case it is a server socket.
6. the application can now process the packet, and either send it using e.g. `csp_send()`, or free the packet using `csp_buffer_free()`.

Routing table

When a packet is routed, the destination address is looked up in the routing table, which results in a `csp_route_t` record. The record contains the inteface (`csp_iface_t`) the packet is to be send on, and an optional `via` address. The `via` address is used, when the sender cannot direcly reach the receiver on one of its connected networks, e.g. sending a packet from the satellite to the ground - the radio will be the `via` address.

CSP comes with 2 routing table implementations (selected at compile time).

- static: supports a one-to-one mapping, meaning routes must be configured per destination address or a single `default` address. The `default` address is used, in case there are no routes set for the specific destination address. The `static` routing table has the fastest lookup, but requires more setup.

- cidr (Classless Inter-Domain Routing): supports a one-to-many mapping, meaning routes can be configued for a range of destianation addresses. The `cidr` is a bit slower for lookup, but simple to setup.

Routes can be configured using text strings in the format:

<address>[/mask] <interface name> [via]

- address: is the destination address, the routing table will match it against the CSP header destination.
- mask (optional): determines how many MSB bits of address are to be matched. mask = 1 will only match the MSB bit, mask = 2 will match 2 MSB bits. Mask values different from 0 and 5, is only supported by the cidr rtable.
- interface name: name of the interface to route the packet on
- via (optional) address: if different from 255, route the packet to the `via` address, instead of the

> address in the CSP header.

Here are some examples:

> - "10 I2C" route destination address 10 to the "I2C" interface and send it to address 10 (no `via`).
> - "10 I2C 30" route destination address 10 to the "I2C" interface and send it to address 30 (`via`). The original destination address 10 is not changed in the CSP header of the packet.
> - "16/1 CAN 4" (CIDR only) route all destinations addresses 16-31 to address 4 on the CAN interface.
> - "0/0 CAN" default route, if no other matching route is found, route packet onto the CAN interface.

Interface

The interface typically implements `layer2`, and uses drivers from `layer1` to send/receive data. The interface is a generic struct, with no knowledge of any specific interface , protocol or driver:

```
1  /**
2     CSP interface.
3  */
4  struct csp_iface_s {
5          uint16_t addr;              // Host address on this subnet
6          uint16_t netmask;           // Subnet mask
7          const char * name;          // Name, max compare length is #CSP_IFLIST_NAME_MAX
8          void * interface_data;      // Interface data, only known/used by the interface lay
9          void * driver_data;         // Driver data, only known/used by the driver layer, e
10         nexthop_t nexthop;          // Next hop (Tx) function
11         uint16_t mtu;               // Maximum Transmission Unit of interface

12         uint8_t split_horizon_off;  // Disable the route-loop prevention
13         uint32_t tx;                // Successfully transmitted packets
14         uint32_t rx;                // Successfully received packets
15         uint32_t tx_error;          // Transmit errors (packets)
16         uint32_t rx_error;          // Receive errors, e.g. too large message
17         uint32_t drop;              // Dropped packets
18         uint32_t autherr;           // Authentication errors (packets)
19         uint32_t frame;             // Frame format errors (packets)
20         uint32_t txbytes;           // Transmitted bytes
21         uint32_t rxbytes;           // Received bytes
22         uint32_t irq;               // Interrupts
23         struct csp_iface_s * next;  // Internal, interfaces are stored in a linked list
24 };
```

If an interface implementation needs to store data, e.g. state information (KISS), it can use the pointer `interface_data` to reference any data structure needed. The driver implementation can use the pointer `driver_data` for storing data, e.g. device number.

See function `csp_can_socketcan_open_and_add_interface()` in

`src/drivers/can/can_socketcan.c` for an example of how to implement a CAN driver and hooking it into CSP, using the CSP standard CAN interface.

Send

When CSP needs to send a packet, it calls `nexthop` on the interface returned by route lookup. If the interface succeeds in sending the packet, it must free the packet. In case of failure, the packet must not be freed by the interface. The original idea was, that the packet could be retried later on, without having to re-create the packet again. However, the current implementation does not yet fully support this as some interfaces modifies header (endian conversion) or data (adding CRC32).

Receive

When receiving data, the driver calls into the interface with the received data, e.g. `csp_can_rx()`. The interface will convert/copy the data into a packet (e.g. by assembling all CAN frames). Once a complete packet is received, the packet is queued for later CSP processing, by calling `csp_qfifo_write()`.

Library Structure

As mentioned in the introduction, CSP is written in C.

There are 5 sub-directories under the libcsp main directory:

- include
- src
- utils
- examples
- doc

The file structure is outlined below:

Folder	Description
`libcsp/include/csp`	Public header files
`libcsp/include/csp/arch`	Architecture (platform)
`libcsp/include/csp/interfaces`	Interfaces
`libcsp/include/csp/drivers`	Drivers
`libcsp/src`	Source modules and internal header files
`libcsp/src/arch`	Architecture (platform) specific code
`libcsp/src/arch/freertos`	FreeRTOS

`libcsp/src/arch/macosx`	Mac OS X
`libcsp/src/arch/posix`	Posix (Linux)
`libcsp/src/arch/windows`	Windows
`libcsp/src/bindings/python`	Python3 wrapper for libcsp
`libcsp/src/crypto`	HMAC, SHA
`libcsp/src/drivers`	Drivers, mostly platform specific (Linux)
`libcsp/src/drivers/can`	CAN
`libcsp/src/drivers/usart`	USART
`libcsp/src/interfaces`	Interfaces, CAN, I2C, KISS, LOOPBACK and ZMQHUB
`libcsp/src/rtable`	Routing tables
`libcsp/src/transport`	Transport layer: UDP, RDP
`libcsp/utils`	Utilities, Python scripts for decoding CSP headers

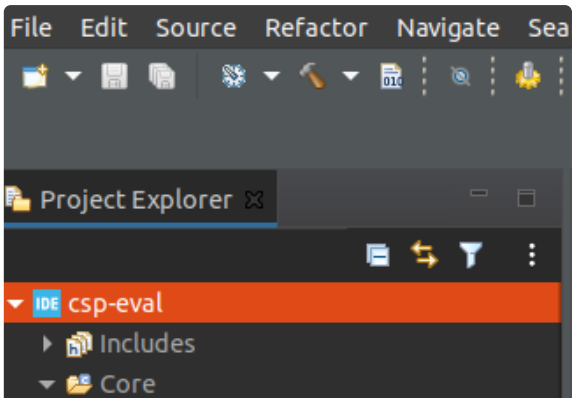
STM32 implementation

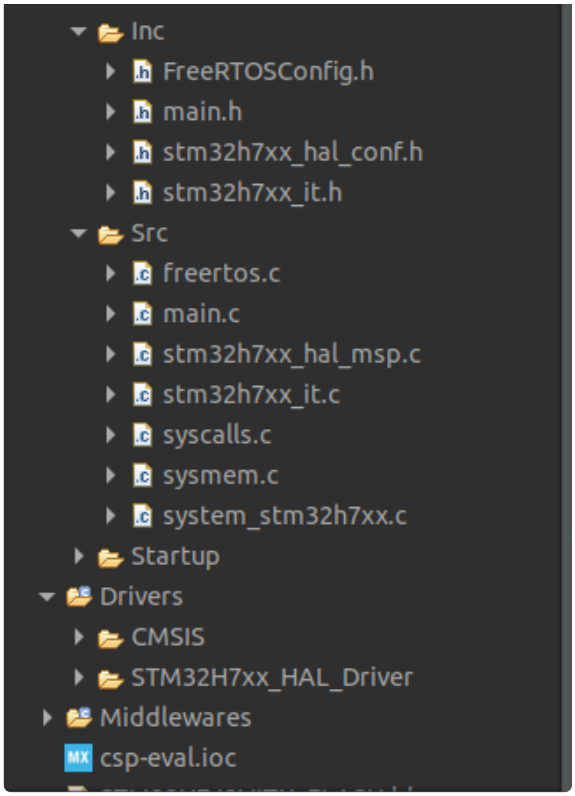
This illustrates the typical application for an ARM-based STM32 microcontroller using the STM Cube IDE. This guide uses RS-485 as the physical layer and FreeRTOS is being used.

Creating a new STM32 project

Create a new STM32 project, select the chip eg STM32H743VITx and use the Device Configuration Tool to set up FreeRTOS with the CMSIS_V2 standard, the RS-485 pins, and any other relevant config.

Save the configuration and allow code generation. Once it is done, your project files should look like this:



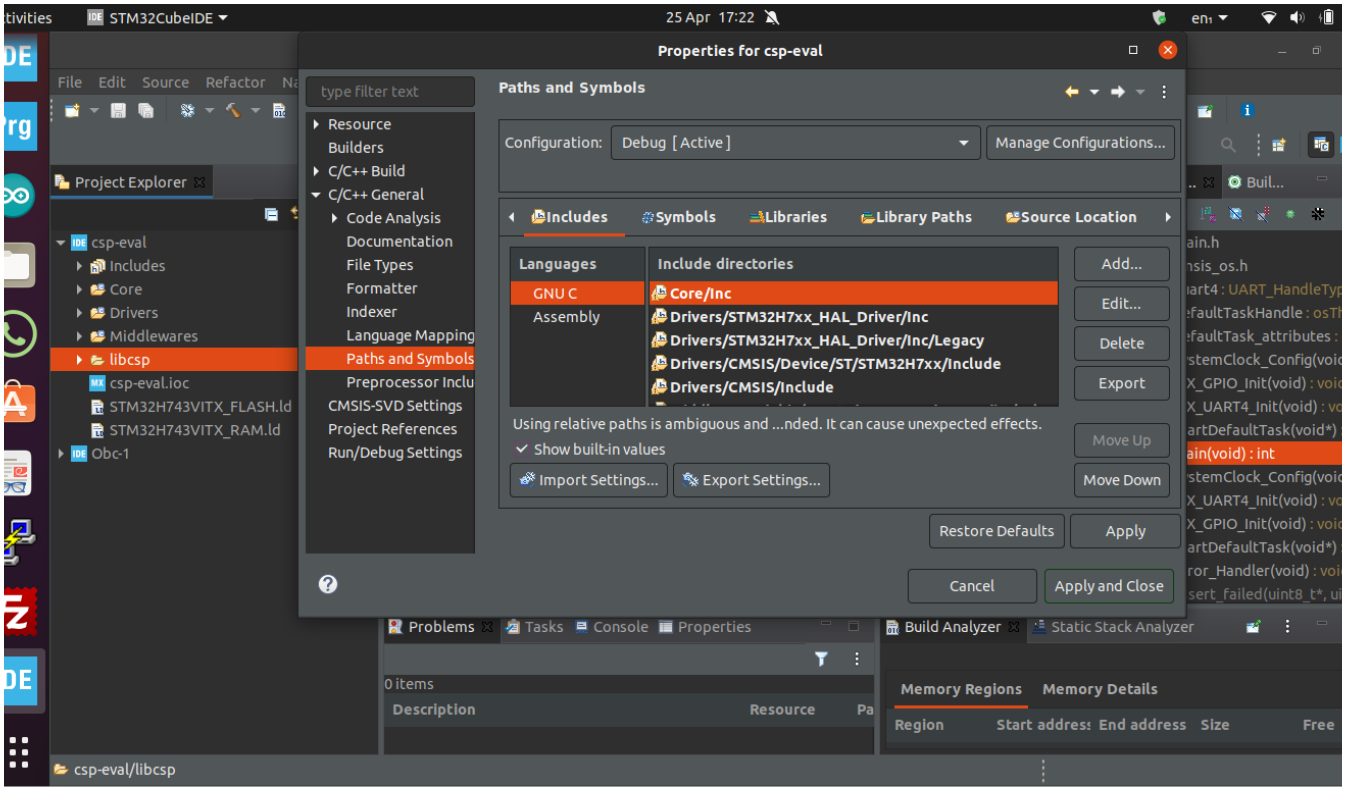


Project tree in STM32 CubeIDE

Download the CSP library source code and include the files in your STM32 CubeIDE workspace as shown below:

First create a new folder and name it 'libcsp' or the designated name that will be used for Tafiti's workflow. Copy the source code of the CSP library into this folder.

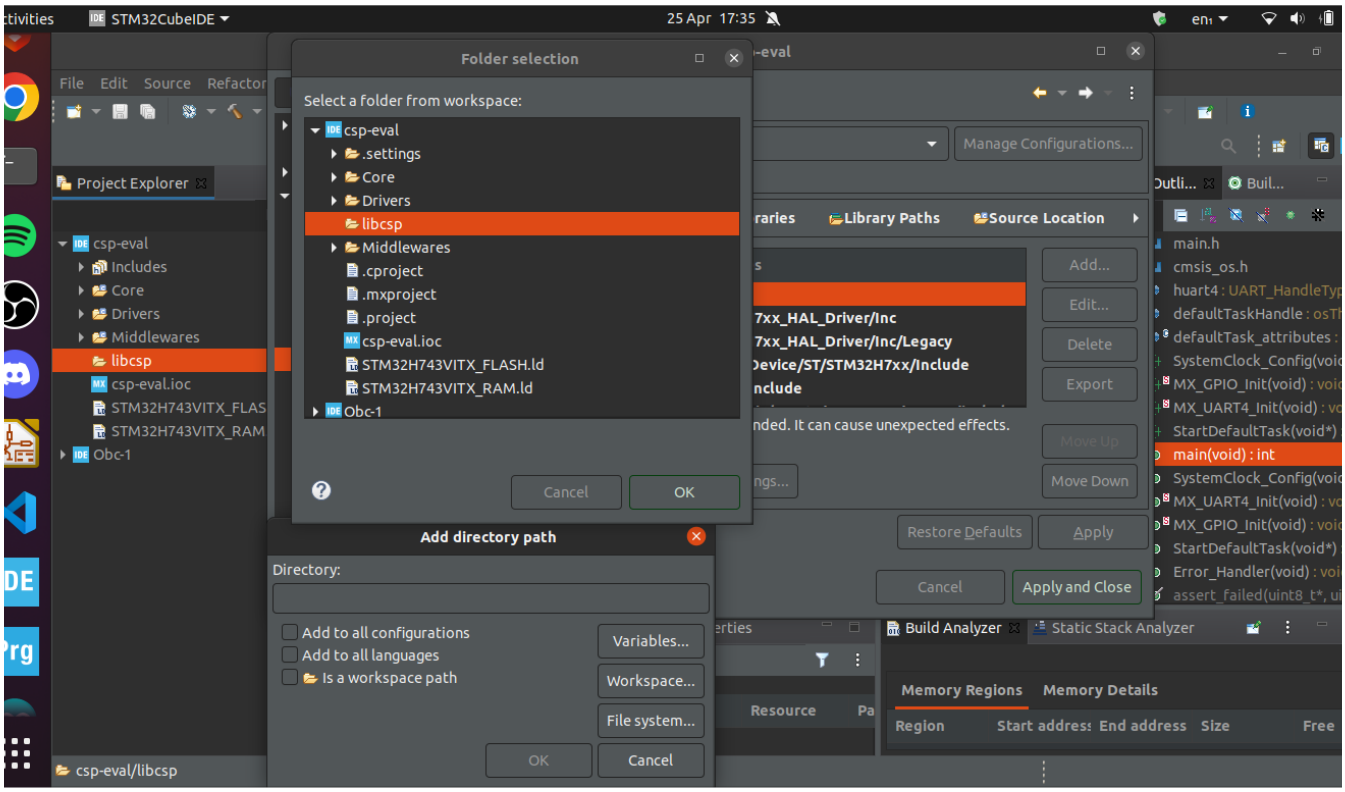
Go to Project -> Properties -> Paths and symbols



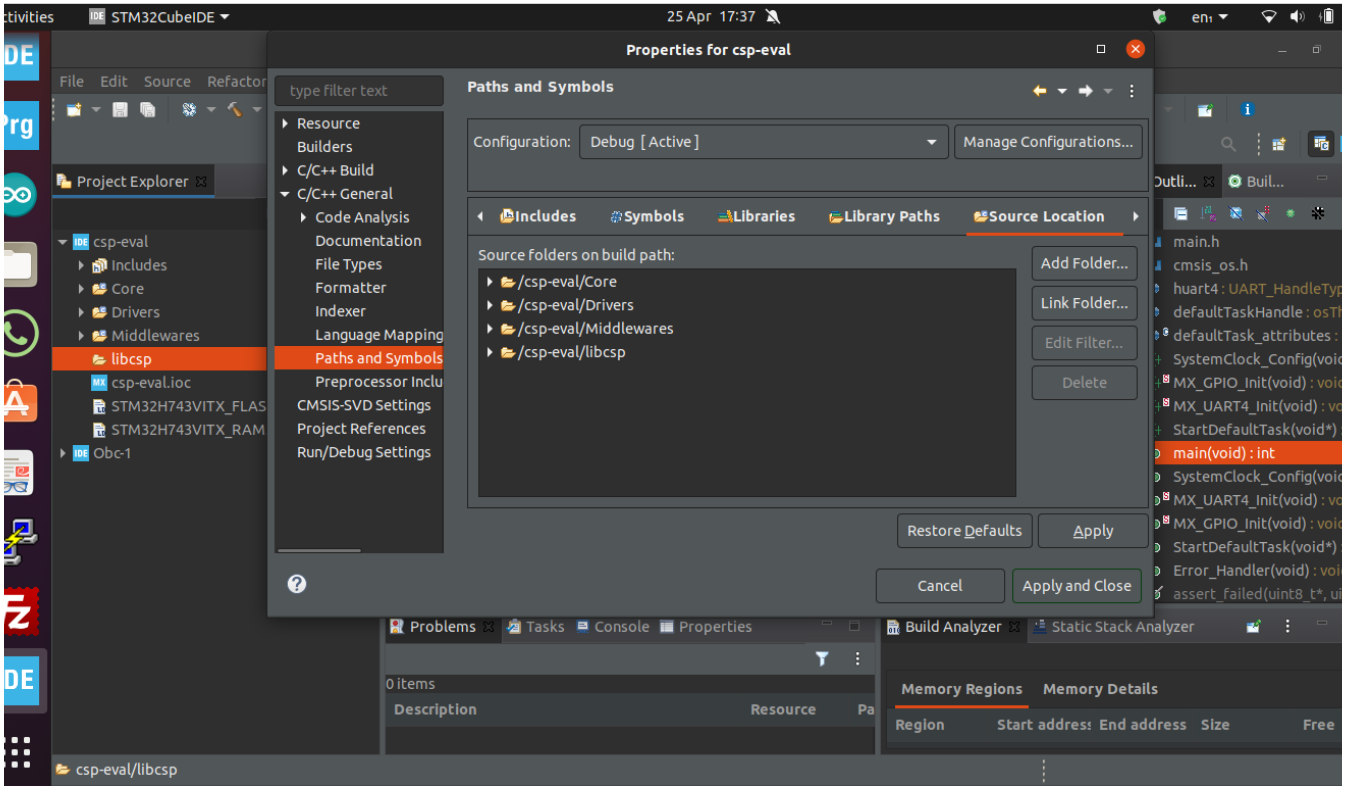
This is where you should be

Under the Includes tab in Paths and Symbols, click add.

Add directory path -> Select libcsp



Do the same for Sources



Now you can incorporate the CSP code into your code base.

More practical examples will be provided.

AVR implementation

This is still under development

Challanges

These are the potential challenges for Tafiti Engineers may encounter when using the CubeSat Space Protocol

1. Significant Workload

CSP was originally developed for the CAN (controller area network) interface. However, it can be used with any standard physical layer, including RS-485. Currently, the developers have extended it to SpaceWire, I2C, and RS-232, but not RS-485.

Using FreeRTOS will provide the necessary drivers, but the interface will have to be developed by the Tafiti engineer.

---

2. Scarcity of examples/tutorials

Unlike the two other technologies proposed by the OBC to be used (FreeRTOS and RS-485) that have plenty of examples and application guides, CSP is not as well documented.

This means that implementation could be an uphill task.